JAVA™ 2

Том I. Основы

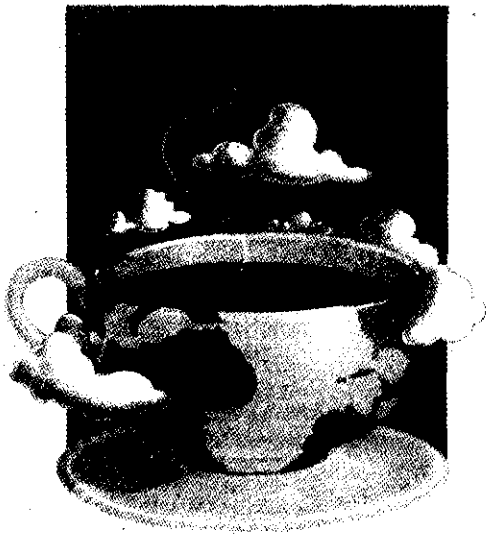
Библиотека профессионала

Седьмое издание

Core JAVA™ 2

Volume I – Fundamentals

Seventh Edition



CAY S. HORSTMANN • GARY CORNELL

Sun Microsystems Press
A Prentice Hall Title

JAVA™ 2

Том I. Основы

Библиотека профессионала

Седьмое издание

КЕЙ С. ХОРСТМАНН • ГАРИ КОРНЕЛЛ

Москва • Санкт-Петербург • Киев
2007

ББК 32.973.26-018.2.75
Х82
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *В.В. Вейтмана*

По общим вопросам обращайтесь в Издательский дом "Вильямс"
по адресу: info@williamspublishing.com, http://www.williamspublishing.com
115419, Москва, а/я 783; 03150, Киев, а/я 152

Хорстманн, Кей С., **Корнелл**, Гари.

Х82 Java 2. Библиотека профессионала, том 1. Основы, 7-е изд. : Пер. с англ. — М. : Издательский дом "Вильямс", 2007. — 896 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-0970-1 (рус.)

Книга адресована прежде всего программистам-профессионалам и представляет собой исчерпывающий справочник и методическое пособие по основам программирования на языке Java. Однако это не просто учебник по синтаксису языка. Назначение книги — обучить методам объектно-ориентированного программирования и научить справляться с основными проблемами в этой области. Работа с книгой не требует опыта программирования на языке C++ и применения методов ООП. Любой программист, работавший с такими языками, как Visual Basic, C, Cobol или Pascal, не будет испытывать затруднений при работе с ней. Книга содержит многочисленные примеры и советы по программированию. Авторы уделили большое внимание возможностям, которые стали доступны программистам с появлением JDK 5.0. Новые языковые и библиотечные средства нашли свое отражение в кодах примеров, приведенных в книге.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Prentice Hall PTR.

Authorized translation from the English language edition published by Prentice Hall PTR, Copyright © 2005 Sun Microsystems, Inc., A Prentice Hall Title

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2007.

ISBN 978-5-8459-0970-1 (рус.)
ISBN 0-13-148202-5 (англ.)

© Издательский дом "Вильямс", 2007
© Sun Microsystems, Inc., 2005

Оглавление

Введение 15

Глава 1. Введение в язык Java 21

Глава 2. Среда программирования на языке Java 37

Глава 3. Основные конструкции языка Java 59

Глава 4. Объекты и классы 127

Глава 5. Наследование 193

Глава 6. Интерфейсы и внутренние классы 265

Глава 7. Программирование графики 305

Глава 8. Обработка событий 349

Глава 9. Компоненты пользовательского интерфейса из пакета Swing 409

Глава 10. Аплеты 583

Глава 11. Исключения и отладка 657

Глава 12. Потоки и файлы 729

Глава 13. Работа с универсальными типами 833

Приложение А. Ключевые слова 867

Приложение Б. Особенности кода, ориентированного на JDK 5.0 869

Предметный указатель 874

Содержание

Введение 15

Благодарности 20

Глава 1

Введение в язык Java **21**

Программная платформа Java 21

Характерные особенности языка Java 22

Простота 22

Поддержка объектов 23

Поддержка распределенных вычислений 24

Надежность 24

Безопасность 24

Независимость от архитектуры компьютера 25

Переносимость 26

Использование интерпретатора 26

Производительность 27

Поддержка потоков 27

Динамичность 28

Java и Internet 28

История языка Java 30

Распространенные заблуждения относительно языка Java 33

Глава 2

Среда программирования на языке Java **37**

Инсталляция Java Development Kit 38

Где найти JDK 38

Установка пути к исполняемым файлам 39

Инсталляция библиотек и документации 40

Установка примеров программ 41

Навигация по каталогам 41

Выбор среды разработки программ 42

Использование инструментов, вызываемых из командной строки 43

Возможные ошибки 45

Использование интегрированной среды разработки программ 46

Сообщения об ошибках компиляции 49

Компиляция и запуск программ из текстового редактора 49

Выполнение графического приложения 52

Создание и запуск аплетов 54

Глава 3

Основные конструкции языка Java 59
Простая программа на языке Java 60
Комментарии 63
Типы данных 63
Целые числа 64
Числа с плавающей точкой 64
Символы 66
Логические значения 67
Переменные 67
Инициализация переменных 69
Константы 69
Операторы 70
Операторы инкрементирования и декрементирования 71
Операторы отношения и логические операторы 72
Побитовые операции 73
Математические функции и константы 74
Преобразование числовых типов 75
Приведение числовых типов 76
Иерархия операторов 76
Нумерованные типы 77
Строки 77
Кодовые точки и кодовые единицы 78
Подстроки 78
Изменение строк 79
Конкатенация 80
Проверка совпадения строк 81
Интерактивная документация по API 84
Ввод и вывод 86
Чтение входных данных 86
Форматирование выходных данных 89
Поток управления 94
Блоки 94
Условные выражения 95
Неопределенные циклы 98
Определенные циклы 102
Многовариантное ветвление — оператор switch 106
Прерывание потока управления 108
Работа с большими числами 110
Массивы 112
Цикл "for each" 113
Инициализация массивов и анонимные массивы 114
Копирование массивов 115
Параметры командной строки 116
Сортировка массива 117
Многомерные массивы 120
"Неровные" массивы 123

Глава 4

Объекты и классы 127
Введение в объектно-ориентированное программирование 127
Основные термины объектно-ориентированного программирования 129
Объекты 130
Отношения между классами 131
Различия между ООП и традиционным процедурным программированием 133
Использование готовых классов 134
Объекты и объектные переменные 135
Класс GregorianCalendar из библиотеки Java 138
Модифицирующие методы и методы доступа 139
Определение собственных классов 145
Класс Employee 145
Использование нескольких исходных файлов 148
Анализ класса Employee 148
Конструкторы 149
Явные и неявные параметры 150
Преимущества инкапсуляции 151
Доступ к данным из различных экземпляров класса 154
Закрытые методы 154
Неизменяемые поля экземпляра 154
Статические поля и методы 155
Статические поля 155
Константы 156
Статические методы 157
Порождающие методы 158
Метод main() 158
Параметры методов 161
Формирование объектов 167
Перегрузка 167
Инициализация полей по умолчанию 167
Конструктор по умолчанию 168
Явная инициализация полей 169
Имена параметров 169
Вызов одного конструктора из другого 170
Инициализационные блоки 171
Уничтожение объекта и метод finalize() 175
Пакеты 175
Импортирование классов 176
Импортирование статических методов и полей 177
Добавление классов в пакеты 178
Как виртуальная машина определяет, где находятся классы 181
Область видимости пакета 184
Комментарии и документирование 185
Включение комментариев 185
Комментарии к классу 186
Комментарии к методам 186

Комментарии к полям 187
Комментарии общего характера 187
Комментарии к пакетам и обзоры 189
Извлечение комментариев 189
Рекомендации по разработке классов 190

Глава 5

Наследование **193**
Классы, суперклассы и подклассы 194
Иерархия наследования 200
Полиморфизм 201
Динамическое связывание 202
Предотвращение наследования: терминальные классы и методы 205
Приведение типов 206
Абстрактные классы 208
Защищенный доступ 213
Object: глобальный суперкласс 214
Метод equals() 215
Проверка эквивалентности объектов и наследование 216
Метод hashCode() 219
Метод toString() 221
Универсальный класс ArrayList 226
Доступ к элементам списочных массивов 229
Совместимость между типизированными и обычными ссылочными массивами 233
Интерфейсные классы и преобразование простых типов 234
Методы с переменным числом параметров 237
Отражение 238
Класс Class 239
Использование отражения для анализа свойств классов 243
Применение отражения для анализа объектов в ходе выполнения программы 248
Применение отражения для работы с универсальным массивом 252
Указатели на методы! 256
Нумерованные типы 260
Как применять наследование 261

Глава 6

Интерфейсы и внутренние классы **265**
Интерфейсы 266
Свойства интерфейсов 271
Интерфейсы и абстрактные классы 273
Клонирование объектов 274
Интерфейсы и обратный вызов 279
Внутренние классы 282
Использование внутренних классов для определения и изменения состояния объекта 284
Специальные синтаксические правила для работы с внутренними классами 287
Чем полезны внутренние классы 288

Локальные внутренние классы 290
Безымянные внутренние классы 293
Статические внутренние классы 295
Proxy-классы 298
Свойства proxy-классов 303

Глава 7

Программирование графики **305**
Общие сведения о пакете Swing 306
Создание фреймов 309
Позиционирование фреймов 312
Отображение информации на панели 318
Двухмерные фигуры 323
Работа с цветом 331
Заполнение фигур 334
Использование шрифтов при выводе текста 336
Дополнительные операции с изображениями 344

Глава 8

Обработка событий **349**
Общие сведения об обработке событий 349
Пример: обработка щелчка на кнопке 352
Использование внутренних классов в качестве обработчиков событий 357
Превращение компонентов в обработчики событий 359
Пример: изменение стиля 361
Пример: обработка событий, связанных с окнами 365
Классы-адаптеры 366
Иерархия событий библиотеки AWT 369
Семантические и низкоуровневые события в библиотеке AWT 371
Взаимосвязь основных понятий, используемых при обработке событий 373
Типы низкоуровневых событий 374
События, связанные с клавиатурой 374
События, связанные с мышью 380
События, связанные с фокусом ввода 388
Действия 392
Многоадресная передача событий 400
Реализация источников событий 403

Глава 9

Компоненты пользовательского интерфейса из пакета Swing **409**
Архитектура "модель–представление–контроллер" 410
Создание интерфейса с помощью схемы ... 410
Работа кнопок из пакета Swing с точки зрения архитектуры "модель–представление–контроллер" 414
Общие сведения о компоновке элементов 415
Диспетчер компоновки BorderLayout 418
Панели 419
Диспетчер компоновки GridLayout 421

Ввод текста 425
Поля редактирования 426
Метки и маркирующие компоненты 428
Отслеживание изменений в полях редактирования 429
Поля для ввода пароля 434
Поля для форматированного ввода 435
Текстовые области 451
Выбор вариантов 455
Флажки опций 455
Переключатели 458
Рамки 462
Раскрывающиеся списки 467
Линейные регуляторы 471
Инкрементный регулятор 477
Меню 486
Создание меню 486
Пиктограммы в пунктах меню 489
Флажки опций и переключатели в меню 490
Контекстные меню 492
Клавиши быстрого доступа и клавишные сокращения 493
Блокирование и разблокирование пунктов меню 496
Панели инструментов 500
Подсказки 502
Расширенные средства компоновки 505
Диспетчер компоновки BoxLayout 508
Заполнители 509
Диспетчер компоновки GridBagLayout 512
Альтернативные способы указания параметров gridx, gridy, gridwidth и gridheight 515
Диспетчер компоновки SpringLayout 524
Компоновка без использования диспетчера 533
Диспетчер компоновки, определяемый разработчиком 534
Порядок обхода компонентов 538
Диалоговые окна 540
Диалоговые окна для выбора вариантов 541
Создание диалоговых окон 553
Обмен данными 557
Диалоговые окна для выбора файлов 564
Диалоговое окно для выбора цвета 576

Глава 10

Аплеты **583**
Основные понятия 584
Простой аплет 585
Выполнение аплета 587
Преобразование приложения в аплет 589
Жизненный цикл аплета 591
Обеспечение безопасности при работе аплета 592

Создание окон с помощью аплетов 594
HTML-дескрипторы и атрибуты для работы с аплетами 596
Атрибуты, задающие расположение аплетов 597
Атрибуты для указания кода аплета 598
Атрибуты для браузеров, поддерживающих Java 600
Дескриптор <object> 601
Передача информации аплетам 601
Мультимедиа 607
Инкапсуляция URL 607
Получение мультимедиа-файлов 607
Контекст аплета 609
Взаимодействие аплетов 609
Отображение элементов в браузере 610
Аплет для поддержки закладок 611
Аплет или приложение? И то и другое! 614
Файлы JAR 620
Упаковка приложений 622
Файл манифеста 622
Самораспаковывающиеся JAR-файлы 623
Ресурсы 624
Герметизация пакета 628
Технология Java Web Start 628
JNLP API 631
Хранение параметров настройки 642
Карты свойств 642
Класс Preferences 650

Глава 11

Исключения и отладка **657**
Обработка ошибок 658
Классификация исключений 659
Объявление контролируемых исключений 661
Генерация исключений 663
Создание классов исключений 664
Перехват исключений 665
Перехват нескольких исключений 667
Повторная генерация исключения 667
Блок finally 668
Анализ стека 671
Обзор обработки ошибок и исключений в языке Java 674
Как использовать исключения 678
Регистрация ошибок 682
Простая регистрация 682
Расширенные средства регистрации 683
Настройка диспетчера регистрации 685
Поддержка регистрационных сообщений на разных языках 686
Обработчики 687
Фильтры 690

Средства форматирования 690
Диагностические утверждения 699
Подключение и отключение диагностических утверждений 700
Рекомендации по использованию диагностических утверждений 701
Приемы отладки 704
Рекомендации по отладке программ 704
Использование консольного окна 710
Отслеживание событий AWT 712
Класс Robot 716
Применение отладчика 720
Отладчик JDB 720
Отладчик Eclipse 725

Глава 12

Потоки и файлы **729**
Потоки 729
Чтение и запись байтов 730
Полное семейство потоков 732
Фильтры потоков 737
Потоки данных 741
Потоки для работы с файлами в режиме произвольного доступа 744
Текстовые потоки 745
Наборы символов 746
Запись текста в поток вывода 753
Чтение текста из потока ввода 756
Потоки, связанные с ZIP-файлами 757
Использование потоков 765
Запись выходных данных с разделителями 766
Класс StringTokenizer и текст с разделителями 767
Чтение текста с разделителями 768
Класс StringBuilder 772
Потоки с произвольным доступом 773
Объектные потоки 778
Сохранение объектов различных типов 779
Формат файла, используемый при сериализации объекта 783
Проблемы, связанные с сохранением ссылок на объекты 786
Формат вывода ссылок на объекты 793
Модификация механизма сериализации, предусмотренного по умолчанию 796
Сериализация перечислений 798
Поддержка версий 799
Применение сериализации для клонирования 802
Работа с файлами 804
Новые средства ввода-вывода 811
Поддержка виртуальной памяти 811
Структура буфера 817
Блокировка файлов 819
Регулярные выражения 821

Глава 13

Работа с универсальными типами **833**
Зачем нужны универсальные типы 834
Кто создает универсальные типы 835
Создание простого универсального класса 836
Универсальные методы 838
Ограничения, накладываемые на переменные типа 839
Универсальный код и виртуальная машина Java 841
Преобразование универсальных выражений 842
Преобразование универсальных методов 843
Обращение к существующему коду 845
Ограничения на работу с универсальными типами 846
Простые типы 846
Запрос типа в процессе выполнения программы 847
Исключения 847
Массивы 848
Создание экземпляров универсальных типов 848
Статический контекст 849
Конфликты в результате приведения к низкоуровневому коду 849
Правила наследования для универсальных типов 850
Подстановочные типы 851
Ограничения супертипов 853
Подстановочные типы без ограничений 856
Перехват подстановочного типа 856
Отражение и универсальные типы 860
Использование параметров Class<T> для проверки типов 861
Информация об универсальных типах в виртуальной машине Java 861

Приложение А

Ключевые слова **867**

Приложение Б

Особенности кода, ориентированного на JDK 5.0 **869**
Модифицированный цикл for 869
Универсальный списочный массив 869
Преобразование простых типов 870
Произвольное число параметров 870
Ковариантные возвращаемые типы 870
Статическое импортирование 870
Консольный ввод 870
Форматированный вывод 871
Делегирование вызовов панели содержимого 871
Кодовые точки Unicode 871
Класс StringBuilder 871

Предметный указатель **874**

Введение

Читателю

Язык Java ворвался в глобальную сеть в конце 1995 года и сразу же завоевал всеобщую популярность. Он обещал стать *универсальным средством*, обеспечивающим связь пользователей с любыми источниками информации, независимо от того, где она расположена — на Web-сервере, в базе данных, справочной системе и т.п. У языка Java есть все предпосылки для этого. Данный язык поддержали все производители программного обеспечения, за исключением компании Microsoft. Присущая Java надежность и наличие средств защиты вселяют уверенность как в программистов, так и в пользователей. Этот язык предоставляет средства, позволяющие решать задачи повышенной сложности, — обеспечивать взаимодействие по сети, управление базами данных и поддерживать выполнение и взаимодействие нескольких потоков.

Начиная с 1995 года компания Sun Microsystems выпустила в свет шесть основных версий набора инструментальных средств Java Development Kit. На протяжении последних девяти лет объем API возрос с 200 до более чем 3000 классов. Теперь он охватывает самые разные области программирования — создание пользовательского интерфейса, управление базами данных, интернационализацию, обеспечение безопасности и представление информации посредством языка XML. Самое большое количество нововведений с момента появления первой реализации было предложено в JDK 5.0, выпущенном в 2004 году.

Книга, которую вы держите в руках, выдержала уже семь изданий. Каждое издание выходило вслед за появлением очередной версии JDK, при этом авторы стремились как можно лучше отразить достоинства новой реализации. В этом издании внимание уделено универсальным наборам данных, новому циклу `for` и другим новшествам, появившимся в JDK 5.0.

Данное издание, как и предыдущие, *рассчитано на опытных программистов, стремящихся использовать язык Java для воплощения реальных проектов*. Мы, как и прежде, обещаем не нервировать читателя и не утомлять его вычурным шрифтом. Предполагается, что читатель этой книги — опытный программист, имеющий солидную подготовку в области языков программирования. *Однако знать язык C++ и объектно-ориентированное программирование совсем не обязательно.* Опираясь на отклики, полученные после опубликования предыдущих изданий, мы по-прежнему убеждены, что программисты, изучавшие языки Visual Basic, C или COBOL, без проблем во всем разберутся. (Читатель может даже совершенно не иметь опыта разработки графического пользовательского интерфейса в средах Windows, Unix или Macintosh.)

От читателя требуется лишь желание создавать реальные программы для решения реальных задач. Такому читателю вряд ли понравятся тривиальные примеры типа перечня кухонного оборудования или фруктовых деревьев.

Книга содержит большое количество программ, демонстрирующих почти все возможности языка и библиотек, обсуждаемые в тексте. Программы преднамеренно упрощены, чтобы читатель сосредоточил внимание именно на рассматриваемых вопросах. Однако в большинстве случаев они являются вполне реальными и не подменяют

настоящее решение задачи его имитацией. Их очень удобно использовать в качестве отправной точки при создании собственных программ.

Предполагается, что читатель стремится овладеть всеми возможностями языка Java. Например, в книге подробно рассматриваются следующие темы.

- Объектно-ориентированное программирование (ООП).
- Отражение и proxy-объекты.
- Интерфейсы и внутренние классы.
- Обработчики событий.
- Разработка графического пользовательского интерфейса с помощью набора инструментальных средств Swing.
- Обработка исключений.
- Потоки ввода-вывода и сериализация объектов.
- Использование универсальных типов.

Из-за большого объема библиотек классов Java невозможно описать в одном томе все средства, необходимые опытным программистам. Поэтому книга разделена на два тома. Первый том, который вы держите в руках, посвящен основным понятиям языка Java, а также основам программирования пользовательского интерфейса. Во втором томе содержится описание особенностей языка Java, используемых при создании профессиональных приложений, и средств программирования пользовательского интерфейса повышенной сложности. В этом томе подробно обсуждаются следующие темы.

- Поддержка нескольких потоков.
- Распределенные объекты.
- Базы данных.
- Расширенные средства создания графического интерфейса.
- Платформенно-ориентированные методы.
- Представление и обработка данных с помощью языка XML.
- Сетевое программирование.
- Классы, описывающие наборы данных.
- Расширенные средства для работы с графикой.
- Интернационализация.
- JavaBeans.

При написании книги неизбежны ошибки и неточности. Мы были бы очень признательны, если бы читатели, обнаружившие их, сообщили нам. Естественно, нам бы хотелось, чтобы поводов для таких сообщений было поменьше. Список наиболее часто задаваемых вопросов, а также исправлений и уточнений можно найти на Web-странице `http://www.horstmann.com/corejava.html`. В конце перечня замеченных ошибок есть анкета, в которой можно указать встреченные неточности и записать свои пожелания. Место для этой анкеты выбрано специально, чтобы читатель сначала прочитал вопросы и ошибки, выявленные другими пользователями. Пожалуйста, не огорчайтесь, если мы не ответим на каждый вопрос или ответим не сразу. Мы читаем все сообщения электронной почты и благодарим вас за стремление сделать будущие издания книги еще понятнее и точнее.

Мы надеемся, что чтение этой книги доставит вам удовольствие и поможет в работе.

О книге

В главе 1 содержится обзор особенностей Java, отличающих его от других языков программирования. Здесь описываются цели, которых стремились достичь разработчики языка, а также результаты, которых они действительно достигли. Далее излагается краткая история создания и развития языка Java.

В главе 2 описан процесс получения и инсталляции пакета JDK. Затем показано, как загрузить примеры программ, рассмотренных в книге. Кроме того, в главе продемонстрирован процесс компиляции и выполнения типичных программ, написанных на языке Java: консольного приложения, графического приложения и аплета. Для работы с данным типом программ использованы пакет JDK, текстовый редактор, в котором предусмотрен вызов компилятора и интегрированная среда Java.

Обсуждение языка Java начинается в главе 3. Здесь описаны основы языка: переменные, циклы и простые функции. Если вы умеете программировать на языках C или C++, то легко поймете, о чем идет речь, поскольку синтаксис Java похож на синтаксис языка C. Если же вы не умеете программировать на языке C, а изучали, например, языки Visual Basic или COBOL, вам следует внимательно прочитать эту главу.

В настоящее время объектно-ориентированное программирование является основным направлением развития программирования вообще, а язык Java представляет собой полностью объектный язык. В главе 4 рассматривается *инкапсуляция* (первое из двух основных понятий объектно-ориентированного программирования) и механизм ее реализации в языке Java, т.е. классы и методы. Кроме правил языка Java, приводятся советы, касающиеся разработки программ с использованием объектного подхода. Наконец, в этой главе описывается замечательная утилита `javadoc`, позволяющая форматировать комментарии в программе в виде совокупности Web-страниц с гипертекстовыми ссылками. Если вы знакомы с языком C++, то можете просто пролистать эту главу. Программисты, не имеющие опыта объектно-ориентированного программирования, должны посвятить некоторое время изучению его основ, прежде чем продолжить изучение языка Java.

Инкапсуляция представляет собой лишь один из принципов объектно-ориентированного программирования. В главе 5 вводится новое понятие – *наследование*, позволяющее программистам использовать существующие классы и модифицировать их для своих нужд. Это – основной способ программирования на языке Java. Механизмы наследования в языках Java и C++ почти одинаковы, поэтому программисты, знающие язык C++, могут сосредоточиться лишь на различиях между этими языками.

В главе 6 показано, как использовать такое понятие языка Java, как *интерфейс*. Интерфейсы позволяют выйти за рамки простого наследования, описанного в главе 5. Освоив интерфейсы, читатель получит возможность полностью использовать всю мощь объектно-ориентированного программирования. Здесь также описаны полезные средства языка Java, называемые *внутренними классами*. Они позволяют делать программы более ясными и четкими.

В главе 7 изучение программирования начинается всерьез. В этой главе показано, как создать окно, нарисовать в нем геометрическую фигуру, отформатировать текст с использованием различных шрифтов и отобразить рисунки.

Глава 8 посвящена обсуждению модели событий AWT (Abstract Windowing Toolkit). Читатель узнает, как написать программу, которая реагировала бы на такие события, как щелчок мыши или нажатие клавиши. Кроме того, в главе показано, как работать

с основными элементами графического пользовательского интерфейса, например с кнопками и панелями.

В главе 9 очень подробно обсуждается набор инструментальных средств Swing, позволяющий создавать графический пользовательский интерфейс, не зависящий от конкретной платформы. Здесь вы узнаете о разнообразных видах кнопок, текстовых компонентах, рамках, линейных регуляторах, списках, меню и диалоговых окнах. Однако некоторые, более сложные компоненты будут рассмотрены во втором томе.

Завершив изучение главы 9, читатель получит представление обо всех механизмах, позволяющих создавать *аплеты*, т.е. небольшие программы, выполняющиеся в среде, реализуемой браузером. Они являются предметом обсуждения главы 10, в которой показаны примеры полезных аплетов. Однако намного важнее то, что аплеты рассматриваются нами как иллюстрация средств *доставки программ* (program deployment). В главе описано, как запаковать приложение в JAR-файл и доставить его по глобальной сети, используя механизм Java Web Start. В заключение объясняется способ, с помощью которого программы, написанные на языке Java, сохраняют и извлекают информацию после своего развертывания.

В главе 11 обсуждается *обработка исключений*, которая позволяет выходить его затруднительных положений, в которые могут попадать даже хорошие программы. Например, при загрузке файла по сети могут произойти разрыв соединения, переполнение диска и другие неприятности. Обработка исключений представляет собой эффективный способ разделения нормального выполнения программы и обработки ошибок. Разумеется, даже если в программе предусмотрены все исключения, остается возможность ее сбоя. Кроме того, в главе приводится большое количество полезных советов, касающихся отладки программ. В заключение рассматриваются примеры отладки программ с помощью утилиты JDB, отладчика интегрированной среды программирования и класса `Robot`.

Глава 12 посвящена вводу-выводу данных. В языке Java любой ввод и вывод выполняется через так называемые *потоки*, позволяющие осуществлять связь с любыми источниками информации, например файлами, узлами сети или блоками памяти. В этой главе подробно описываются классы, предназначенные для ввода и вывода, с помощью которых легко работать с форматом Unicode, а также раскрывается механизм сериализации объектов. В заключение описывается несколько библиотек, включенных в версию пакета SDK 1.4: новые классы ввода-вывода, поддерживающие более сложные и эффективные операции над файлами, а также библиотеки регулярных выражений.

Завершается книга обзором способов программирования с использованием *универсальных типов*, которые были реализованы в JDK 5.0. Универсальные типы упрощают восприятие кода программы и позволяют добиться более надежной работы. Мы покажем, как избежать лишнего приведения типов, снижающего надежность программы.

В приложении А перечислены ключевые слова языка Java.

В приложении Б рассказано, как модифицировать код, чтобы он мог быть обработан компиляторами из предыдущих версий JDK (например, версии 1.4).

Принятые соглашения

Как и в большинстве книг по программированию, тексты программ представляются с помощью моноширинного шрифта.

В книге много замечаний, касающихся языка C++. Они описывают различия между языками Java и C++. Если у вас нет опыта программирования на языке C++ или вы вспоминаете о нем, как о кошмаре, который хочется поскорее забыть, можете пропустить эти замечания.

Так оформляются обычные замечания.

Так выглядят советы разработчику.

Так мы предупреждаем о возможной опасности.

Язык Java сопровождается большой библиотекой, реализующей прикладной программный интерфейс (Application Programming Interface — API). Каждое обращение к API сопровождается кратким описанием, которое отмечается пиктограммой и размещается в конце соответствующего раздела. Эти описания не формальны, однако они воспринимаются проще, чем официальная документация по API. Описание классов API сопровождается номером версии, в которой впервые данный класс был реализован. Это поможет пользователям легче ориентироваться в версиях библиотеки.

Листинги программ оформляются следующим образом.

Листинг 2.4. Содержимое файла `WelcomeApplet.java`

Исходные тексты программ

На Web-странице `http://www.phptr.com/corejava` содержатся все исходные тексты программ, рассмотренных в книге. Файлы находятся в сжатом виде. Их можно распаковать либо с помощью хорошо известной программы `unzip`, либо утилиты `jar`, являющейся частью пакета Java Development Kit. Более подробная информация об инсталляции этого пакета и соответствующие примеры приведены в главе 2.

Благодарности

Писать книгу всегда трудно. Создавать ее переработанный вариант ненамного легче, особенно если учесть постоянное изменение технологии программирования. В процесс издания книги всегда вовлечено много специалистов, и мне доставляет большое удовольствие выразить благодарность всем им.

Сотрудники Prentice Hall PTR и Sun Microsystems Press оказали нам неоценимую помощь, однако их имен нет на обложке книги. Я хотел бы выразить им всем большую благодарность. Как всегда, я горячо благодарю своего редактора Грега Донча (Greg Doench), работающего в издательстве Prenbtice-Hall PTR, за работу, выполненную им в процессе создания и публикации рукописи, а также за то, что он привлек к работе многих специалистов. Приношу благодарность своему соавтору по предыдущим изданиям книги Гари Корнеллу (Gary Cornell), который в настоящее время занят другими проектами.

Благодарю многочисленных читателей, сообщивших мне о досадных ошибках и предложивших, как улучшить книгу. Я особенно благодарен рецензентам, детально изучившим рукопись и предотвратившим появление еще большего количества ошибок.

Рецензентами этого и предыдущих изданий были: Чак Эллисон (Chuck Allison), редактор *C/C++ Users Journal*; Алек Битон (Alec Beaton), компания PointBase, Inc.; Дэвид Браун (Dawid Brown); д-р Николас Дж. Де Лилло (Dr. Nicholas J. De Lillo) колледж Манхэттена; Ракеш Дхупар (Rakesh Dhoopar), компания Oracle; Дэвид Гери (David Geary), компания Sabreware; Анджела Гордон (Angela Gordon) и Дэн Гордон (Dan Gordon), компания Sun Microsystems; Роб Гордон (Rob Gordon), Камерон Грегори (Cameron Gregory), Марти Холл (Marty Hall), лаборатория прикладной физики университета Джонса Хопкинса (The Johns Hopkins University Applied Physics Lab); Винсент Харди (Vincent Hardy), компания Sun Microsystems; Владимир Иванович (Vladimir Ivanovic), компания PointBase, Inc.; Джерри Джексон (Jerry Jackson), компания ChannelPoint Software; Тим Киммет (Tim Kimmet), компания Preview Systems; Крис Лаффра (Chris Laffra), Чарли Лай (Charlie Lai), компания Sun Microsystems; Дуг Лэнгстон (Doug Langston), Дуг Ли (Doug Lea), компания SUNY Oswego; Грэгори Лонгшо (Gregory Longshore), Боб Линч (Bob Lynch), Марк Моррисси (Mark Morrissey), Oregon Graduate Institute; Махеш Нилаканта (Mahesh Neelakanta), Атлантический университет Флориды; Пол Фильон (Paul Philion), Блэйк Рагсделл (Blake Ragsdell), Стюарт Реджес (Stuart Reges), университет Аризоны; Питер Сандер (Peter Sander), университет ESSI (Ницца, Франция); Пол Севинк (Paul Sevinc), TeamUp AG; Деванг Шах (Devang Shah), компания Sun Microsystems; Брэдли Э. Смит (Bradley A. Smith), Кристофер Тейлор (Christopher Taylor), Люк Тейлор (Luke Taylor), компания Valtech; Джордж Тируватукал (George Thiruvathukal), Ким Топли (Kim Topley) (автор *Core JFC*), Джанет Трауб (Janet Traub), Питер ван дер Линден (Peter van der Linden), компания Sun Microsystems; Барт Уолш (Burt Walsh), Корки Картрайт (Corky Cartwright), Френк Коен (Frank Cohen), компания PushToTest; Крис Крейн (Chris Crane), компания devXsolution; Билл Хиггинс (Bill Higgins), компания IBM; Хэнг Лью (Hang Lau), университет МакГилл; Анджелика Ленгер (Angelica Langer), Марк Лоуренс (Mark Lawrence), д-р Пол Сангера (Paul Sanghera) и Стив Стелтинг (Steve Stelting), компания Sun Microsystems.

Кей Хорстманн
Сан Франциско, 2004.

Глава 1

Введение в язык Java

В этой главе...

- Программная платформа Java
- Характерные особенности языка Java
- Java и Internet
- История языка Java
- *Распространенные заблуждения относительно языка Java*

На появление первой версии Java в 1996 году откликнулись не только специализированные компьютерные газеты и журналы, но даже такие издания, как *The New York Times*, *The Washington Post* и *Business Week*. Java – единственный язык программирования, удостоившийся десятиминутного репортажа на National Public Radio. Это было удивительное время. Тем дням и последующей истории языка Java посвящена данная глава.

Программная платформа Java

В первой редакции этой книги говорилось следующее.

> *"Как язык программирования Java перевыполнил рекламные обещания. Несомненно, это один из лучших языков, доступных серьезным программистам.*
> *Потенциально Java имел все предпосылки, чтобы стать великим языком программирования, однако, вероятно, сейчас время для этого уже упущено. Когда появляется новый язык, немедленно возникает проблема его совместимости с программным обеспечением, созданным ранее".*

По поводу этого абзаца наш редактор долго спорил с одним из руководителей компании Sun. Но и сейчас, по прошествии долгого времени, такая оценка кажется нам правильной. Действительно, Java обладает целым рядом преимуществ, о которых мы поговорим чуть позже. Однако более поздние дополнения далеко не так элегантны, как исходный вариант языка, и виной тому пресловутые требования совместимости.

Как уже говорилось в первой редакции, Java — не только язык. Хорошие языки — не редкость; появление некоторых из них вызвало в свое время сенсацию в компьютерном мире. В отличие от них, Java — это программная платформа, включающая мощную библиотеку, большой объем кода, вполне пригодного для повторного использования, и среду для выполнения программ, которая обеспечивает безопасность, независимость от операционной системы и автоматическую "сборку мусора".

Программистам нужны языки с четкими синтаксическими правилами и понятной семантикой. Такому требованию, помимо Java, отвечают десятки языков. Некоторые из них даже обеспечивают переносимость и "сборку мусора", однако их библиотеки оставляют желать лучшего. В результате программисты вынуждены самостоятельно реализовывать графические операции, действия с базой данных и другие часто встречающиеся процедуры. Java объединяет в себе прекрасный язык, высококачественную среду выполнения программ и обширную библиотеку. В результате многие программисты остановили свой выбор именно на Java.

Характерные особенности языка Java

Авторы Java написали руководство, в котором объяснялись цели и достоинства нового языка. В этом документе приведено одиннадцать характерных особенностей Java. Этот язык:

простой;
объектно-ориентированный;
распределенный;
надежный;
безопасный;
не зависит от архитектуры компьютера;
переносимый;
интерпретируемый;
высокопроизводительный;
многопотоковый;
динамичный.

В данном разделе мы

- приведем цитаты из руководства по языку Java, раскрывающие особенности языка;
- поделимся с читателями соображениями по поводу отдельных свойств языка, основываясь на собственном опыте работы с его последней версией.

Руководство, о котором идет речь, можно найти по адресу `http://java.sun.com/docs/white/langenv/`. Характерные особенности Java описаны в документе, который находится по адресу `ftp://ftp.javasoft.com/docs/papers/java-overview.ps`.

Простота

"Мы хотели создать систему, которая позволяла бы легко писать программы, не требовала дополнительного обучения и учитывала сложившуюся практику и стандарты программирования. Мы считали C++ не совсем подходящим для этих целей, однако, чтобы сделать систему более доступной, язык Java был разработан максимально похожим на него. Исключили мы лишь редко используемые, малопонятные и невразумительные средства C++, которые, по нашему мнению, приносят больше вреда, чем пользы".

Синтаксис языка Java, по существу, представляет собой упрощенный вариант синтаксиса языка C++. В этом языке нет файлов заголовков, указателей структур, объединений, перегрузки операторов, виртуальных базовых классов и т.п. (Различия между Java и C++ упоминаются на протяжении всей книги.) Однако разработчики не стремились исправить все недостатки языка C++. Например, синтаксис оператора `switch` в языке Java остался неизменным. Зная C++, несложно перейти к языку Java.

Если вы привыкли к визуальной среде программирования (например, Visual Basic), язык Java может показаться сложным. Его синтаксис некоторые считают странным (хотя понять смысл выражения не составляет труда). И что самое главное, при работе на языке Java приходится намного больше программировать. Начинающие программисты нередко предпочитают Visual Basic, так как его визуальная среда программирования позволяет почти автоматически создавать инфраструктуру приложения. Чтобы достичь того же результата с помощью языка Java, необходимо вручную написать достаточно большой объем кода, но программы, получающиеся при этом, более компактны.

> *"Другой аспект простоты – краткость. Одна из целей языка Java – обеспечить разработку независимых программ, способных выполняться на машинах с ограниченным объемом ресурсов. Размер основного интерпретатора и средств поддержки классов составляет около 40 Кбайт; стандартные библиотеки и средства поддержки потоков, в том числе автономное микроядро (self-contained microkernel), занимают еще 175 Кбайт".*

Это огромный успех. Заметим, однако, что библиотеки, предназначенные для поддержки графического пользовательского интерфейса, имеют значительно больший объем.

Поддержка объектов

> *"По сути, объектно-ориентированное программирование – это способ разработки, в центре внимания которого находятся данные (т.е. объекты) и средства доступа к ним. Проводя аналогию со столярным делом, можно сказать, что "объектно-ориентированный" мастер в основном сосредоточен на стуле, который он изготавливает, и лишь во вторую очередь его интересуют инструменты, необходимые для этого; в то же время "не объектно-ориентированный" столяр думает лишь о своих инструментах. Средства языков Java и C++, предназначенные для работы с объектами, по существу совпадают".*

За прошедшие 30 лет объектный подход уже доказал свое право на жизнь, и без него невозможно представить себе современный язык программирования. Действительно, особенности Java, связанные с объектами, сравнимы с языком C++. Основное различие между ними заключается в механизме множественного наследования, который в языке Java заменен механизмом интерфейсов. Механизмы отражения (см. главу 5) и сериализации (см. главу 12) позволяют реализовать постоянные (persistent) объекты и средства для создания графических пользовательских интерфейсов на основе готовых компонентов.

Если вы никогда не использовали объектно-ориентированные языки, внимательно изучите главы 4–6. В них излагаются основы объектного подхода и демонстрируются его преимущества над такими традиционными, процедурно-ориентированными языками, как C или Basic, проявляющиеся при разработке сложных проектов.

Поддержка распределенных вычислений

> *"Язык Java предоставляет разработчику обширную библиотеку программ для передачи данных на базе протоколов TCP/IP (Transmission Control Protocol/Internet Protocol – протокол управления передачей/Internet-протокол), HTTP (Hypertext Transfer Protocol – протокол передачи гипертекстовой информации) и FTP (File Transfer Protocol – протокол передачи файлов). Приложения, написанные на языке Java, могут открывать объекты и получать к ним доступ по сети; при этом для адресации используются URL (Uniform Resource Location – унифицированный локатор ресурсов)".*

Язык Java предоставляет мощные и удобные средства для работы в сети. Каждый, кто когда-либо пытался писать программы для работы в глобальной сети, будет приятно удивлен тем, как легко решаются на Java самые трудные задачи, например открытие гнезд (socket). (Работа в сети будет описана во втором томе данной книги.) Связь между распределенными объектами в языке Java обеспечивается механизмом вызова удаленных методов (эта тема также затрагивается во втором томе).

В настоящее время сложные распределенные приложения обычно реализуются на базе Java 2 Enterprise Edition (J2EE).

Надежность

> *"Язык Java предназначен для написания программ, которые должны надежно работать в любых ситуациях. Основное внимание в данном языке уделяется раннему обнаружению возможных ошибок контролю в процессе выполнения программы, а также устранению ситуаций, которые могут вызвать ошибки... Единственное существенное отличие языка Java от языка C++ заключается в модели указателей, принятой в Java, которая исключает возможность записи в произвольно выбранную область памяти и повреждения данных".*

Компилятор языка Java выявляет такие ошибки, которые в других языках обнаруживаются только на этапе выполнения программы. Кроме того, программисты, потратившие многие часы на поиски ошибки, вызвавшей повреждение памяти из-за неверного указателя, будут очень рады тому, что при работе с Java такие проблемы возникнуть в принципе не могут.

Если раньше вы программировали на Visual Basic либо на другом языке, в котором указатели явным образом не используются, возможно, вам будет непонятно, почему этот вопрос настолько важен. Программистам, использующим язык C, нужны указатели для доступа к строкам, массивам, объектам и даже файлам. При программировании на Visual Basic ничего этого не требуется, и разработчик может не беспокоиться о распределении памяти для хранения объектов. С другой стороны, многие структуры данных в языке, не имеющем указателей, реализовать очень трудно. Для простых конструкций, вроде строк и массивов, указатели не нужны. Возможности указателей в полной мере проявляются лишь там, где без них нельзя обойтись, например, при создании связных списков. Программист на языке Java навсегда избавлен от неверных указателей, опасности неправильного распределения памяти и ее некорректного использования.

Безопасность

> *"Язык Java предназначен для использования в сетевой или распределенной среде. По этой причине большое внимание было уделено безопасности. Java позволяет создавать системы, защищенные от вирусов и несанкционированного доступа".*

В первом издании мы написали: "Никогда не говори *никогда*", — и оказались правы. Вскоре после выхода первой версии Java Development Kit группа экспертов по вопросам безопасности из Принстонского университета обнаружила первые ошибки в системе защиты Java 1.0. Компания Sun Microsystems развернула исследования в области безопасности Java-программ. В частности, она обнародовала спецификацию и код виртуальной машины и библиотек, ответственных за защиту. Это ускорило выявление и устранение ошибок. Следует заметить, что количество таких ошибок было относительно невелико и чтобы воспользоваться ими, злоумышленник должен был бы обладать довольно глубокими знаниями.

Перечислим некоторые ситуации, возникновение которых предотвращает система безопасности языка Java.

- Намеренное переполнение стека выполняемой программы — один из распространенных способов атаки.
- Повреждение участков памяти, находящихся за пределами пространства, выделенного процессу.
- Несанкционированное чтение файлов и их модификация.

Со временем в язык были добавлены новые средства защиты. В версии 1.1 в Java появилось понятие классов с цифровой подписью (см. том 2). Пользуясь таким классом, вы получаете сведения об авторе. Если вы доверяете автору, то можете предоставить этому классу все необходимые привилегии.

Альтернативный механизм доставки кода, предложенный компанией Microsoft, опирается на технологию ActiveX и для безопасности использует только цифровые подписи. Очевидно, что этого недостаточно — любой пользователь программного обеспечения компании Microsoft может подтвердить, что даже программы широко известных производителей часто завершаются аварийно, создавая тем самым опасность повреждения данных. Система безопасности в языке Java намного надежнее, чем технология ActiveX, поскольку она контролирует приложение в процессе его работы.

Независимость от архитектуры компьютера

"Компилятор генерирует объектный файл, формат которого не зависит от архитектуры компьютера. Скомпилированная программа может выполняться на любых процессорах; для ее работы необходима лишь исполняющая система Java. Код, генерируемый компилятором Java, называется байтовым кодом. Он разработан таким образом, чтобы на любой машине его можно было легко интерпретировать либо в процессе работы перевести в набор команд, ориентированных на конкретный процессор".

Эта идея не нова. Более двадцати лет назад она была предложена Никлаусом Виртом (Niclaus Wirth) для языка Pascal; эта же технология была реализована в системе UCSD Pascal.

Очевидно, что код, интерпретируемый с помощью виртуальной машины, всегда будет работать медленнее, чем машинные инструкции, поэтому целесообразность такого подхода у многих вызывает сомнение. Однако эффективность байтового кода можно существенно повысить за счет компиляции, осуществляемой в процессе выполнения программы. Этот механизм доказал свою эффективность и даже использовался при создании платформы .NET.

Следует отметить, что виртуальная машина имеет ряд преимуществ по сравнению с непосредственным выполнением программы. Она существенно повышает безопасность, поскольку в процессе работы можно оценить последствия выполнения каждой конкретной команды. Некоторые программы даже способны динамически генерировать байтовый код в соответствии с текущими потребностями.

Переносимость

> *"В отличие от языков C и C++, ни один из элементов спецификации Java не зависит от реализации. И размер основных типов данных, и арифметические операции над ними строго определены".*

Например, тип `int` в языке Java всегда означает 32-разрядное целое число. В языках C и C++ тип `int` может означать как 16-разрядное, так и 32-разрядное целое число. Единственное ограничение заключается в том, что размер типа `int` не может быть меньше размера типа `short int` и больше размера типа `long int`. Фиксированный размер числовых типов позволяет избежать многих неприятностей, связанных с выполнением программ на разных компьютерах. Бинарные данные хранятся и передаются в неизменном формате, что также позволяет избежать недоразумений, связанных с разным порядком следования байтов на разных платформах. Строки сохраняются в стандартном формате Unicode.

> *"Библиотеки, представляющие собой часть системы, предоставляют интерфейс, обеспечивающий переносимость программ. Например, в языке предусмотрен абстрактный класс `Window` и его реализации для операционных систем Unix, Windows и Macintosh".*

Каждый, кто когда-либо пытался написать программу, которая одинаково хорошо работала бы под управлением операционных систем Windows, Macintosh и десятка разновидностей системы Unix, знает, что это очень трудная задача. Разработчики версии Java 1.0 предприняли героическую попытку решить эту проблему, предоставив простой инструментальный набор, адаптирующий обычные элементы пользовательского интерфейса к различным платформам. К сожалению, библиотека, на которую было затрачено немало труда, не позволила достичь приемлемых результатов в разных системах. (В результате для каждой системы в графических программах проявлялись свои характерные ошибки.) Однако это было лишь началом. Во многих приложениях машинная независимость оказалась намного важнее изысканности графического пользовательского интерфейса. Именно эти приложения выиграли от появления версии Java 1.0. Однако теперь инструментальный набор для создания графического пользовательского интерфейса полностью переработан и больше не зависит от интерфейсных средств, используемых на конкретном компьютере. Новая версия более осмысленна и, по нашему мнению, более привлекательна для пользователя, чем предыдущие.

Использование интерпретатора

> *"Java-программа может выполняться на любой машине, для которой реализован интерпретатор. Поскольку редактирование связей – относительно легкий процесс, разработка программ может стать намного быстрее и эффективнее".*

Инкрементное связывание имеет свои преимущества, однако в данном случае оценка несомненно завышена. В любом случае средства разработки Java работают довольно медленно. Если вы, например, привыкли к скорости, которую обеспечивает среда Microsoft Visual C++, то, перейдя на Java, вы поначалу будете разочарованы. (Последняя версия Visual Studio существенно проигрывает по сравнению с классической средой разработки. Независимо от того, каким языком вы пользуетесь, вам явно придется обратиться к руководству с просьбой выделить более быстродействующий компьютер.)

Производительность

> *"Хотя обычно интерпретируемые байтовые коды имеют достаточную производительность, бывают ситуации, когда требуется еще более высокая эффективность. Байтовые коды можно транслировать во время выполнения программы в машинные команды для процессора, на котором выполняется данное приложение".*

Если для выполнения байтовых кодов применяется интерпретатор, не следует употреблять словосочетание "высокая производительность". Однако на многих платформах возможен специальный тип компиляции, обеспечиваемый так называемыми синхронными компиляторами (just-in-time compilers – JIT). Они транслируют байтовые коды в машинориентированные команды, сохраняют результат в памяти, а затем по мере необходимости вызывают его. Поскольку при этом интерпретация выполняется только один раз, данный подход во много раз увеличивает скорость работы. Несмотря на то что синхронные компиляторы не обеспечивают такой производительности, как компиляторы, ориентированные на конкретные типы процессоров, они работают намного быстрее интерпретаторов, обеспечивая для некоторых программ 10- и даже 20-кратное ускорение. Эта технология постоянно совершенствуется, и не исключено, что со временем она обеспечит скорость, характерную для кода, генерируемого традиционными компиляторами. Например, синхронный компилятор может определить, какой фрагмент кода выполняется чаще, и оптимизировать его по скорости выполнения.

Поддержка потоков

> *"Наличие нескольких потоков обеспечивает лучшую интерактивность и возможность выполнения программ в реальном масштабе времени".*

Если вы когда-либо пытались организовать многопотоковые вычисления на каком-нибудь из языков программирования, вы будете приятно удивлены тем, как это легко сделать на Java. Потоки в языке Java могут использовать преимущества многопроцессорных систем, если операционная система позволяет это сделать. К сожалению, реализации потоков на большинстве систем сильно отличаются друг от друга, а разработчики языка Java не предпринимают должных усилий, чтобы достичь независимости от платформы и в этом вопросе. Только код для вызова потоков остается одинаковым для всех машин, а ответственность за их поддержку возлагается на операционную систему или библиотеку потоков. (Потоки описываются во втором томе.) Несмотря на это, именно простота организации многопотоковых вычислений делает язык Java таким привлекательным для разработки программного обеспечения серверов.

Динамичность

> *"Во многих отношениях язык Java является более динамичным, чем языки C и C++. Он был разработан так, чтобы легко адаптироваться к постоянно изменяющейся среде. В библиотеки можно свободно включать новые методы и объекты, никак не затрагивая этим приложения, пользующиеся библиотеками. Язык Java позволяет легко получать информацию о ходе выполнения программы".*

Это очень важно в тех случаях, когда требуется добавить код в уже выполняемую программу. Ярким примером этого является код, который загружается из глобальной сети для выполнения браузером. В версии Java 1.0 получить информацию о работающей программе было непросто, однако последняя версия языка Java раскрывает перед программистом как структуру, так и поведение объектов. Это весьма ценно для систем, которые должны анализировать объекты в ходе выполнения программы. К таким системам относятся средства создания графического пользовательского интерфейса, интеллектуальные отладчики, включаемые компоненты и базы данных.

Компания Microsoft выпустила продукт под названием J++, который очень похож на Java. Синтаксис языка практически совпадает с Java. Подобно Java, J++ использует для выполнения байтовых кодов виртуальную машину, однако интерфейс с внешним кодом существенно отличается. Кроме того, дополнительные языковые конструкции, реализованные Microsoft, вряд ли пригодны для чего-то, кроме взаимодействия с Windows API. Основные библиотеки Java и J++ (используемые для работы со строками, с сетевыми соединениями, для поддержки потоков, выполнения математических вычислений и т.д.) совершенно идентичны. В то же время библиотеки для работы с графикой, создания пользовательских интерфейсов и доступа к удаленным объектам существенно отличаются друг от друга. В настоящее время Microsoft прекратила поддержку J++ и сосредоточила свои усилия на другом языке, который называется C# и несколько напоминает Java, но использует другую виртуальную машину. Создан даже язык J#, предназначенный для переноса приложений J++ на виртуальную машину, используемую C#. Языки J++, C# и J# в данной книге рассматриваться не будут.

Java и Internet

При работе по сети Java-программы используются просто — пользователи загружают байтовые коды языка Java по сети и выполняют их на своих машинах. Программы Java, работающие под управлением Web-браузеров, называются *аплетами*. Для использования аплета нужен Web-браузер, поддерживающий язык Java и способный интерпретировать байтовые коды. Лицензия на исходные коды языка Java принадлежит компании Sun, настаивающей на неизменности как самого языка, так и структуры его основных библиотек. К сожалению, в реальности все не так. Разные версии браузеров Netscape и Internet Explorer поддерживают разные версии языка Java, причем некоторые из этих версий значительно устарели. Эта неприятная ситуация создает все больше препятствий при разработке аплетов, которые позволяли бы использовать преимущества последней версии языка Java. Чтобы решить данную проблему, компания Sun разработала программу *Java Plug-in*, позволяющую формировать наиболее современную среду для запуска Java-программ в браузерах Netscape и Internet Explorer (подробнее об этом — в главе 10).

Загрузка аплета напоминает вставку рисунка на Web-страницу. Аплет становится частью страницы, а текст обтекает занимаемое им пространство. Изображение, реализуемое аплетом становится динамичным. Оно реагирует на команды пользователя, изменяет в зависимости от них свой внешний вид и выполняет обмен данными между

компьютером, на котором выполняется аплет, и компьютером, где этот аплет постоянно хранится.

На рис. 1.1 показан хороший пример динамической Web-страницы, выполняющей *сложные вычисления и применяющей аплет для отображения моделей молекул.* Чтобы лучше понять структуру молекулы, можно вращать ее либо изменять масштаб изображения, используя мышь. Подобные эффекты нельзя реализовать на статических Web-страницах, однако аплеты делают это возможным. (Этот аплет можно найти по адресу `http://jmol.sourceforge.net`.)

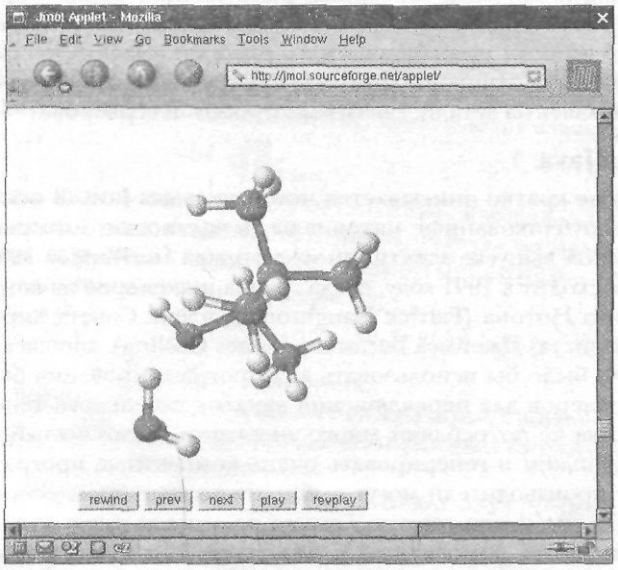


Рис. 1.1. Аплет Jmol

С помощью аплетов на Web-страницу можно добавлять новые кнопки и текстовые поля. Однако, если соединение производится по телефонной линии, такие аплеты медленно загружаются. Почти то же самое можно сделать с помощью языка Dynamic HTML, форм языка HTML (Hypertext Markup Language – язык разметки гипертекстовых данных) или языка сценариев, например JavaScript. Первые аплеты предназначались для анимации: вращающиеся глобусы, танцующие персонажи мультфильмов, изменяющиеся тексты и т.п. Однако для решения этой задачи требовалась лишь малая часть возможностей аплетов: большинство из перечисленного могут делать и анимированные GIF-файлы. Поэтому аплеты целесообразно применять не столько для оформления Web-страниц, сколько для реализации сложных взаимодействий с пользователем.

Несовместимость браузеров и медленная загрузка аплетов по телефонным соединениям привели к тому, что аплеты не стали широко распространенным элементом оформления Web-страниц. В сетях intranet ситуация совершенно иная. В них обычно не возникают проблемы, связанные с пропускной способностью канала, поэтому время загрузки аплетов оказывается очень небольшим. Пользователи intranet обычно применяют одинаковые браузеры, а при необходимости можно применять Java Plug-In. Сотрудники не могут неверно установить программу, реализованную в виде аплета, а системному администратору не нужно обходить все клиентские машины и обновлять на них программное обеспечение. На предприятиях многие программы, предназначенные для учета товаров,

планирования отпусков, возмещения транспортных расходов и т.п., часто используются именно в виде аплетов, выполняющихся под управлением браузеров.

За время работы над книгой внимание многих разработчиков переместилось от клиентских программ к серверным приложениям. В частности, *серверы приложений* (application servers) могут использовать возможности виртуальной машины языка Java для автоматического выравнивания нагрузки, составления пулов соединений с базой данных, синхронизации объектов, обеспечения безопасного завершения работы и повторной загрузки, а также для выполнения других процессов, необходимых для функционирования масштабируемых серверных приложений. Таким образом, программисты, создающие приложения, получили в свое распоряжение готовые механизмы и избавлены от необходимости разрабатывать их самостоятельно. Это повысило производительность труда разработчиков – они могут сосредоточить внимание на самих приложениях, не отвлекаясь на детали, связанные с работой серверов.

История языка Java

В данном разделе кратко описывается история языка Java. В основу раздела положены различные опубликованные материалы (в частности, интервью с создателями языка Java в июльском выпуске электронного журнала *SunWorld* за 1995 год).

История Java восходит к 1991 году, когда группа инженеров из компании Sun под руководством Патрика Нотона (Patrick Naughton) и члена Совета директоров (и разностороннего специалиста) Джеймса Гослинга (James Gosling) занялась разработкой языка, который можно было бы использовать для программирования бытовых устройств, например, контроллеров для переключения каналов кабельного телевидения. Так как подобные устройства не потребляют много энергии и имеют малый памяти, язык должен был быть маленьким и генерировать очень компактные программы. Кроме того, поскольку разные производители могут выбирать разные процессоры, было важно не привязаться к конкретной архитектуре. Проект получил кодовое название "Green".

Стремясь реализовать компактный и независимый от платформы код, разработчики возродили модель, использованную при реализации первых версий языка Pascal на заре эры персональных компьютеров. Никлаус Вирт, создатель языка Pascal, в свое время разработал переносимый язык, генерирующий промежуточный код для некоей гипотетической машины. Такие гипотетические машины называются *виртуальными*, отсюда термин – виртуальная машина Java (Java virtual machine – JVM). Этот промежуточный код можно выполнять на любой машине, имеющей соответствующий интерпретатор. Инженеры, работавшие над проектом "Green", также использовали виртуальную машину, что решило их основную проблему.

Большинство сотрудников компании Sun имели опыт работы с операционной системой Unix, поэтому в основу разрабатываемого ими языка был положен язык C++, а не Pascal. Они сделали язык объектно-, а не процедурно-ориентированным. Как сказал Гослинг в своем интервью: "Язык — это всегда средство, а не цель". Сначала Гослинг решил назвать его Oak (Дуб). (Возможно потому, что он любил смотреть на дуб, растущий прямо под окнами его офиса в компании Sun.) Потом сотрудники компании узнали, что слово Oak уже используется в качестве имени ранее созданного языка программирования, и изменили название на Java.

В 1992 году в рамках проекта Green был выпущен первый продукт, названный *7. Это было устройство интеллектуального дистанционного управления. (Помещаясь в объеме 6×4×4 дюйма, оно имело мощность рабочей станции SPARK.) К сожалению, ни одна из компаний – производителей электронной техники не заинтересовалась этой разработкой. Затем группа стала заниматься созданием устройства для кабельно-

го телевидения, которое могло бы осуществлять новые виды услуг, например включать видеосистему по требованию. И снова они не получили ни одного контракта. Примечательно, что одной из компаний, отказавшихся подписать с ними контракт, руководил Джим Кларк (Jim Clark) – основатель компании Netscape, впоследствии сделавшей очень много для развития языка Java.

Весь 1993 год и половину 1994 года продолжались безрезультатные поиски покупателей продукции, разработанной в рамках проекта "Green" (группа, работающая над проектом, выделилась под названием "First Person, Inc."). Патрик Нотон, один из основателей группы, в основном занимавшийся маркетингом, налетал в общей сложности более 300 тысяч миль, пытаясь продать разработанную технологию. Работа группы "First Person, Inc." была прекращена в 1994 году.

Тем временем в рамках глобальной сети развивалась система World Wide Web. Ключевым элементом этой системы является браузер, превращающий гипертекстовые данные в изображение на экране. В 1994 году большинство пользователей применяли некоммерческий Web-браузер Mosaic, разработанный в суперкомпьютерном центре Университета штата Иллинойс в 1993 году. Частично этот браузер был написан Марком Андреессеном (Mark Andreessen). В то время Марк заканчивал университет, и браузер был его дипломной работой. (Затем он стал одним из основателей и ведущим специалистом компании Netscape, и к нему пришли слава и богатство.)

В своем интервью журналу *Sun World* Гослинг сказал, что в середине 1994 года разработчики языка поняли: "Нам нужно создать высококачественный браузер. Такой браузер должен представлять собой приложение, соответствующее технологии клиент-сервер, в которой жизненно важным является именно то, что мы сделали: архитектурная независимость, выполнение в реальном времени, надежность, безопасность – вопросы, которые не были чрезвычайно важны для рабочих станций. И мы создали такой браузер".

Реальный браузер был разработан Патриком Нотоном и Джонатаном Пэйном (Johnatan Payne). Позднее он был доработан и получил имя HotJava. Чтобы продемонстрировать все возможности Java, браузер был написан на этом языке. Однако разработчики не забывали о таких средствах, которые теперь называются аплетами, наделив свой продукт способностью выполнять код внутри Web-страниц. Продукт, созданный в рамках новой технологии, был представлен на выставке SunWorld '95 23 мая 1995 года и вызвал всеобщий интерес к Java.

Компания Sun выпустила первую версию Java в начале 1996 года. Через несколько месяцев после нее появилась версия Java 1.02. Пользователи быстро поняли, что версия Java 1.02 не подходит для разработки серьезных приложений. Конечно, эту версию можно применять для реализации визуальных эффектов на Web-страницах, например располагать произвольным образом элементы текста, однако версия Java 1.02 была еще сырой. В ней даже отсутствовали средства вывода на принтер. В следующей версии, Java 1.1, были устранены наиболее очевидные недостатки, намного улучшены средства отражения и реализована новая модель событий для программирования графического пользовательского интерфейса. Несмотря на это, ее возможности все еще были ограничены.

Выпуск версии Java 1.2 стал основной новостью конференции JavaOne в 1998 году. В новой версии слабые средства для создания графического пользовательского интерфейса и графических приложений были заменены мощным инструментарием. Это был шаг вперед, к реализации лозунга "Write Once, Run Anywhere" ("Однажды реализовано — везде выполняется"), выдвинутого при разработке предыдущих вер-

сий. В декабре 1998 года через три дня (!) после выхода в свет название новой версии было изменено на громоздкое словосочетание *Java 2 Standard Edition Software Development Kit Version 1.2* (Стандартная редакция пакета инструментальных средств для разработки программного обеспечения на языке Java 2, версия 1.2).

Кроме Standard Edition, были предложены еще два варианта: Micro Edition ("микроредакция") для портативных устройств, например для мобильных телефонов, и Enterprise Edition (редакция для корпоративных приложений). В нашей книге внимание в основном уделено Standard Edition.

Версии 1.3 и 1.4 пакета инструментальных средств Standard Edition намного совершеннее первоначальной реализации Java 2. Они обладают новыми возможностями, производительность их повышена, и, разумеется, они содержат намного меньше ошибок. В процессе развития Java многие взгляды на аплеты и серверные программы были пересмотрены. В частности, оказалось, что на базе Java можно разрабатывать высококачественные серверные приложения.

В версии 5.0 язык Java подвергся наиболее существенной модификации с момента реализации версии 1.1. (Первоначально версия 5.0 имела номер 1.5, но на конференции JavaOne в 2004 г. была принята новая нумерация версий.) После многолетних исследований были добавлены новые базовые конструкции, хотя при этом не были выдвинуты требования модификации виртуальной машины. Ряд элементов, например циклы "for each", автоматическое представление простых типов в виде объектов и метаданные были явно "навеяны" языком C#. Изменение и дополнение базовых языковых конструкций – болезненный процесс, но многие из средств версии 5.0 настолько удобны, что разработчики наверняка примут их с энтузиазмом. В табл. 1.1 приведена информация об этапах развития *языка* Java.

В табл. 1.2 показан рост объема библиотеки API по мере появления новых версий пакета JDK.

Таблица 1.1. Этапы развития языка Java

Версия	Новые языковые средства
1.0	Реализация языка
1.1	Внутренние классы
1.2	Отсутствуют
1.3	Отсутствуют
1.4	Директивы
5.0	Универсальные классы, цикл "for each", представление простых типов в виде объектов, метаданные, нумерация, статическое импортирование

Таблица 1.2. Рост объема библиотеки API для Java Standart Edition

Версия	Количество классов и интерфейсов
1.0	211
1.1	477
1.2	1524
1.3	1840
1.4	2723
5.0	3270

Распространенные заблуждения относительно языка Java

В этом разделе перечислены и прокомментированы некоторые из распространенных заблуждений, касающихся языка Java.

Язык Java – это расширение языка HTML.

Java — это язык программирования, а язык HTML — это способ описания структуры Web-страниц. Между ними нет ничего общего, за исключением HTML-дескрипторов, позволяющих размещать на Web-страницах аплеты, написанные на языке Java.

Я использую XML, поэтому мне не нужен язык Java.

Java — это язык программирования, а XML — средство описания данных. Данные, описанные с помощью языка XML, можно обрабатывать программами, написанными на любом языке программирования, но лишь библиотека API языка Java содержит превосходные средства для обработки таких данных. Кроме того, на языке Java реализованы многие программы независимых производителей, предназначенные для работы с XML-документами. Более подробную информацию по этому вопросу можно найти во втором томе данной книги.

Язык Java легко выучить.

Нет ни одного языка программирования, сопоставимого по возможностям с Java, который можно было бы легко выучить. Простейшие программы написать несложно, однако выполнять серьезную работу всегда трудно. Кроме того, обратите внимание на то, что в этой книге обсуждению собственно языка Java посвящены только четыре главы. В остальных главах рассматривается работа с библиотеками, содержащими тысячи классов и интерфейсов, а также многие тысячи методов. Конечно, невозможно выучить наизусть все классы, однако для выполнения реального проекта пользователю необходимо ориентироваться в содержимом библиотек.

Язык Java со временем станет универсальным языком программирования для всех платформ.

Теоретически это возможно. Именно об этом мечтают все производители программного обеспечения, кроме компании Microsoft. Однако есть много приложений, прекрасно работающих на персональных компьютерах, которые не будут так же хорошо работать на других устройствах или под управлением браузера. Кроме того, эти приложения написаны так, чтобы максимально использовать возможности процессоров и платформенно-ориентированных библиотек. В любом случае они уже перенесены на все важные платформы. К таким приложениям относятся текстовые и графические редакторы, а также Web-браузеры. Обычно эти приложения пишутся на языке C или C++, и пользователь ничего не выиграет, если переписать их на языке Java. Помимо всего прочего, после реализации на языке Java эти программы станут, по крайней мере, в ближайшем будущем, менее эффективными.

Язык Java – это не более чем еще один язык программирования.

Java — прекрасный язык программирования. Многие программисты отдают предпочтение именно ему, а не языку C или C++. Однако в мире существуют сотни великолепных языков, так и не получивших широкого распространения, в то время как языки с очевидными недостатками, такие как C++ и Visual Basic, достигли ошеломительных успехов.

Почему так происходит? Успех любого языка программирования в основном определяется его *системой поддержки*, а не элегантностью его синтаксиса. Существуют ли стандартные библиотеки, позволяющие сделать именно то, что необходимо про-

граммисту? Разработана ли удобная среда для создания и отладки программ? Интегрирован ли язык и его инструментарий в инфраструктуру компьютера? Язык Java достиг успехов в области серверных приложений, поскольку его библиотеки классов позволяют легко сделать то, что раньше было трудно реализовать, например, поддерживать работу в сети и обеспечивать выполнение нескольких потоков. Тот факт, что язык Java уменьшил количество ошибок, связанных с указателями, также говорит в его пользу. Благодаря этому производительность труда программистов повысилась. Однако не в этом кроется причина его успеха.

С появлением языка C# Java устарел.

Разработчики C# использовали многие удачные технические решения, применявшиеся в языке Java, такие как полностью определенный язык, использование виртуальной машины и систему "сборки мусора". Однако по ряду характеристик C# уступает Java, это вопросы безопасности и переносимости программ. Самое очевидное преимущество C# – это прекрасная среда разработки. Если вы работаете в системе Windows, может быть, вам стоит использовать C#. Если же оценить реальную ситуацию, то на сегодняшний день Java остается основным инструментом очень многих программистов.

Java разработан и поддерживается компанией Sun, поэтому пользоваться им не следует.

Мнения на этот счет расходятся. Когда недостатки данного языка были очевидны, хотелось, чтобы их поскорее исправили в рамках открытой системы. Однако на сегодняшний день ситуация не так проста.

Несмотря на то что Sun сохраняет полный контроль над Java, в работу над данным языком и его библиотеками уже оказались вовлечены многие другие компании. Исходных код виртуальной машины и библиотек доступен всем желающим, но его можно использовать лишь для ознакомления, вносить в него изменения запрещено.

Если оценить известные на данный момент открытые языки, ясно видно, что они никак не превосходят Java. Наиболее популярными считаются "три буквы P" в "LAMP" (Linux, Apache, MySQL и Perl/PHP/Python). Эти языки имеют ряд положительных качеств, но в то же время в них не согласованы изменения, вносимые в разные версии, возможности библиотек ограничены, а возможности инструментов разработки явно недостаточны.

Отдельного рассмотрения заслуживают C++ и C#, выработкой стандартов для которых занимались независимые комитеты. Нельзя не согласиться, что их рекомендации более "прозрачны" по сравнению с деятельностью Java Community. Но на практике оказывается, что применимость выработанных спецификаций ограничена. Недостаточно стандартизовать язык и большинство основных библиотек. В реальных проектах очень скоро приходится выходить за рамки действий по обработке строк, наборов данных и файлов. Поэтому, если проанализировать C#, оказывается, что для основной массы библиотек стандартов попросту не существует.

Не исключено, что в будущем Java станет открытым языком, но чтобы это произошло, к такому мнению должны прийти многие специалисты, которые в настоящее время отвечают за развитие Java.

Программы на языке Java работают под управлением интерпретатора, значит, серьезные приложения будут выполняться слишком медленно.

В начале своего развития Java-программы действительно интерпретировались. Теперь в состав всех виртуальных машин, за исключением "микроредакции", входит

синхронный компилятор. Основные элементы кода будут выполняться не медленнее, чем если бы они были написаны на C++.

Для Java характерны дополнительные накладные расходы по сравнению с C++, которые, впрочем, не имеют никакого отношения к использованию виртуальной машины. Они связаны с процедурой "сборки мусора". Автоматическая "сборка мусора", конечно же, происходит несколько медленнее, чем управление памятью вручную, и для Java-программ требуется немного больше памяти, чем для программ, выполняющих аналогичные функции, но написанных на C++. Запуск программы – достаточно длительная процедура, в особенности для очень больших приложений. Пользовательский интерфейс имеет меньшее быстродействие, так как его элементы являются независимыми от конкретной платформы.

Годами пользователи жалуются на низкое быстродействие Java по сравнению с C++. Однако за это время мощность компьютеров многократно возросла, и не исключено, что на новых машинах Java-приложения будут выполняться быстрее, чем работали на старом оборудовании программы, написанные на C++. Теперь, когда нет оснований жаловаться на низкое быстродействие, некоторые противники Java утверждают, что качество интерфейсных элементов, реализуемые средствами данного языка, очень низкое.

Все программы на языке Java выполняются под управлением Web-браузеров.

Все *аплеты*, написанные на языке Java, действительно выполняются под управлением Web-браузеров. Собственно, это и есть определение аплета – программа, выполняемая в среде, реализуемой Web-браузером. Однако вполне возможно и уместно создавать независимые Java-программы, которые никак не связаны с Web-браузером. Эти программы (обычно называемые *приложениями*) являются полностью переносимыми. Их можно без всяких изменений выполнять на другой машине! Поскольку язык Java более удобен и менее подвержен ошибкам, чем язык C++, многие разработчики выбирают именно его. А если учесть наличие средств доступа к базам данных, например Java Database Connectivity (см. том 2), то трудно предпочесть Java какой-то другой язык. Очень удобен Java для начинающих, выбравших его в качестве первого языка программирования.

Большинство программ, рассматриваемых в этой книге, являются независимыми приложениями. Конечно, аплеты очень забавны. Однако Java-приложения более важны и полезны.

Java-аплеты представляют большую опасность для системы защиты.

В свое время было опубликовано несколько отчетов об ошибках в системе защиты языка Java. Большинство из них касалось реализаций языка Java в конкретных браузерах. Исследователи восприняли это как вызов и принялись искать глобальные недостатки в системе защиты Java, которые делали бы бессмысленной модель безопасности аплетов. Их усилия не увенчались успехом. Обнаруженные ошибки в конкретных реализациях вскоре были исправлены, и, после этого, насколько мы знаем, ни одна реальная система не была взломана посредством аплета. Чтобы оценить значение этого факта, вспомните о миллионах вирусных атак на выполняемые файлы операционной системы Windows и макросы редактора Word, действительно вызвавшие немало бед. При этом критика недостатков платформы была удивительно беззуба. Кроме того, механизм ActiveX в браузере Internet Explorer мог бы вызвать много нареканий, однако способы его взлома настолько очевидны, что лишь немногие специалисты потрудились опубликовать результаты своих исследований.

Некоторые системные администраторы даже стали выключать системы защиты языка Java в своих браузерах, чтобы пользователи могли, как и прежде, загружать исполняемые файлы, элементы управления ActiveX и документы, созданные с помощью текстового процессора Word. Можно утверждать, что в настоящее время вероятность преодоления системы защиты аплетов на языке Java сравнима с вероятностью погибнуть в авиационной катастрофе, в то время как риск заразить компьютер вирусом, открыв документ, созданный текстовым процессором Word, куда больше, чем риск погибнуть под колесами автомобиля, перебегая дорогу в час пик.

Язык JavaScript – упрощенная версия языка Java.

Язык JavaScript — это язык сценариев, который можно *использовать в составе Web-страницы.* Он был разработан компанией Netscape и сначала назывался LiveScript. Синтаксис языка JavaScript напоминает синтаксис языка Java, однако на этом их сходство заканчивается (за исключением имени, конечно). Подмножество JavaScript было стандартизовано как ECMA-262, однако его расширения, необходимые для реальной работы, стандартизованы не были. В результате сложный JavaScript-сценарий, который можно было бы выполнять как под управлением браузеров компании Netscape, так и с помощью браузера Internet Explorer, остался несбыточной мечтой.

Пользуясь языком Java, можно заменить компьютер Internet-устройством стоимостью 500 долларов.

Некоторые специалисты уверены, что это непременно случится. Нам кажется абсурдным думать, что пользователи, работающие дома, решат заменить мощные и удобные персональные компьютеры устройствами без жестких дисков с ограниченными возможностями. Однако сетевой компьютер, оснащенный средствами языка Java, предоставляет вполне реальную возможность реализовать "нулевую административную инициативу", направленную на уменьшение стоимости компьютеров, занятых в бизнесе.

Мы рассматриваем Internet-устройство как *дополнение* к персональному компьютеру. Если цена позволяет, почему бы не приобрести специальную аппаратуру для просмотра на экране электронной почты и новостей? Ядро языка Java настолько мало, что он является очевидным выбором для телефонов и других подобных аппаратов.

Ответы на часто задаваемые вопросы о языке Java можно найти в документе по адресу `http://www.apl.jhu.edu/~hall/java/FAQs-and-Tutorials.html`.

Глава 2

Среда программирования на языке Java

В этой главе...

- Инсталляция Java Development Kit
- Выбор среды разработки программ
- Использование инструментов, вызываемых из командной строки
- Использование интегрированной среды разработки программ
- Компиляция и запуск программ из текстового редактора
- Выполнение графического приложения
- Создание и запуск аплетов

В этой главе вы научитесь инсталлировать Java Development Kit (JDK), а также компилировать и запускать на выполнение программы различных типов: консольные программы, графические приложения и аплеты. Мы будем использовать средства пакета JDK, набирая команды в окне, предназначенном для работы с оболочкой. Однако многие программисты предпочитают интегрированные среды разработки. Вы узнаете, как использовать широко распространенную интегрированную среду разработки программ для компиляции и выполнения программ, написанных на языке Java. Изучать интегрированные среды разработки программ и пользоваться ими довольно легко, однако они долго загружаются и предъявляют высокие требования к ресурсам компьютера, так что применять их для разработки небольших программ не имеет смысла. В качестве компромисса можно использовать текстовый редактор, из которого вызываются компилятор и интерпретатор языка Java. Овладев приемами, рассмотренными в этой главе, и выбрав инструмент для разработки программ, вы сможете перейти к главе 3, с которой и начинается изучение языка Java.

Инсталляция Java Development Kit

Наиболее полные и современные версии Java 2 Standart Edition (J2SE) реализованы для операционных систем Solaris, Linux и Windows. Версии для Macintosh и многих других платформ лицензированы и поставляются производителями соответствующих систем.

Где найти JDK

Если вам необходимо скопировать на свой компьютер Java Development Kit, вам надо обратиться на Web-страницу Sun и, приложив немалые усилия, чтобы разобраться в терминах, найти требуемое программное обеспечение.

Аббревиатура JDK вам уже знакома. Она, как нетрудно догадаться, означает Java Development Kit. Некоторые трудности может вызвать тот факт, что в версиях 1.2–1.4 пакет называется Java SDK (Software Development Kit). Кое-где вы найдете и привычный термин JDK.

Далее вам надо найти последовательность символов J2SE, которая означает Java 2 Standard Edition. Существуют также J2EE (Java 2 Enterprise Edition) и J2ME (Java 2 Micro Edition).

Термин "Java 2" был введен в 1998 году, когда специалисты компании Sun по маркетингу поняли, что очередной номер реализации никак не отражает глобальных отличий между JDK 1.2 и предыдущими версиями. Но, поскольку такое мнение сформировалось уже после выхода в свет JDK, было решено, что номер версии 1.2 останется за инструментальным пакетом. Последующие реализации JDK имеют номера 1.3, 1.4 и 5.0. Платформа же была переименована из Java в Java 2. Таким образом, последний инструментальный пакет называется Java 2 Standard Edition Development Kit version 5.0, *или J2SE 5.0.*

Для нас, инженеров, эти манипуляции с именами версий выглядят, по меньшей мере, странно. Именно поэтому мы никогда не сможем достичь успеха в маркетинге.

Если вы используете операционную систему Solaris, Linux или Windows, можете загрузить Java Development Kit, обратившись по адресу `http://java.sun.com/j2se`. Выберите версию 5.0 или более позднюю реализацию и укажите требуемую платформу.

Время от времени компания Sun выпускает наборы, содержащие, помимо Java Development Kit, интегрированную среду разработки. Это интегрированное окружение в разные периоды времени называлось Forte, Sun ONE Studio, Sun Java Studio и Netbeans. Трудно предугадать, какое имя оно будет носить в тот момент, когда вы обратитесь на Web-узел компании Sun. На данном этапе нам нужен только Java Development Kit. Если впоследствии вы решите поработать с интегрированной средой разработки, то сможете скопировать ее с сервера `http://netbeans.org`.

После того как JDK окажется на вашем компьютере, следуйте инструкциям по инсталляции для вашей платформы. На момент написания данной книги инструкции были представлены в документе `http://java.sun.com/j2se/5.0/install.html`.

От конкретной системы зависит только процесс инсталляции и компиляции. После того как вы убедились в работоспособности Java, вы сможете непосредственно выполнять рекомендации, приведенные в данной книге. Независимость от операционной системы является одним из главных преимуществ Java перед многими другими языками.

Процедура инсталляции предлагает по умолчанию имя каталога для установки JDK. В имя каталога входит номер версии, например `jdk5.0`. Это может показаться не столь важным, но данный подход упрощает работу с новыми версиями JDK, установленными для проверки.

Если вы работаете в системе Windows, мы рекомендуем не принимать предлагаемый каталог. Обычно это `c:\Program Files\jdk5.0`. По крайней мере, откажитесь от `Program Files` и разместите пакет в другой позиции файловой системы.

В данной книге мы используем для ссылки на каталог, содержащий Java Development Kit, имя *jdk*. Например, если в тексте указано имя *jdk*`/bin`, значит, речь идет о каталоге `/usr/local/jdk5.0/bin` или `c:\jdk5.0\bin`.

Установка пути к исполняемым файлам

После инсталляции JDK вам нужно сделать еще один шаг: добавить имя каталога *jdk*`/bin` в список путей, по которым операционная система ищет исполняемые файлы. В различных системах это действие выполняется по-разному.

- В системе Unix (включая Solaris и Linux) процедура редактирования путей к выполняемым файлам зависит от используемой оболочки. В случае C shell (эта оболочка принимается на платформе Solaris по умолчанию) добавьте в конец файла `~/.cshrc` следующую строку:

```
set path=(/usr/local/jdk/bin $path)
```

- Если вы работаете с оболочкой Bourne Again (она принимается по умолчанию на платформе Linux), добавьте в конец файла `~/.bashrc` или `~/.bash_profile` строку, приведенную ниже.

```
export PATH=/usr/local/jdk/bin:$PATH
```

- Для других оболочек системы Unix нужно найти способ выполнить аналогичную процедуру.
- На платформе Windows 95/98/Me добавьте строку, приведенную ниже, в файл `AUTOEXEC.BAT`:

```
SET PATH=c:\jdk\bin;%PATH%
```

- Заметьте, что вокруг знака равенства нет пробелов. Чтобы эффект от включения этой строки проявился, нужно перезагрузить компьютер.
- На платформе Windows NT/2000 откройте панель управления, выберите пиктограмму System, а затем вкладку Environment. Прокрутите содержимое окна User Variables, пока не найдете переменную `PATH`. Добавьте имя каталога *jdk*`/bin` в начало списка; новый элемент списка отделяется от уже существующих точкой с запятой, например:

```
c:\jdk\bin; остальное
```

- Сохраните выполненные установки. В любом новом консольном окне заданный путь будет использоваться для поиска выполняемых файлов.

Правильность установок можно проверить следующим образом.

Откройте окно оболочки. Как вы это сделаете, зависит от операционной системы. Введите следующую строку:

```
java -version
```

Нажмите клавишу <Enter>. На экране должно появиться следующее сообщение:

```
java version "5.0"
Java (TM) 2 Runtime Environment, Standard Edition
Java HotSpot (TM) Client VM
```

Если вместо этого сообщения появится строка наподобие "`java:command not found`", "`Bad command or file name`" или "`The name specified is nor recognized as an internal or external command, operable program or batch file`", нужно еще раз проверить, правильно ли выполнена инсталляция и установлен путь.

Инсталляция библиотек и документации

Библиотечные файлы поставляются в пакете JDK в виде архива `src.zip`. Вам следует распаковать этот файл, чтобы получить доступ к исходным текстам программ. Мы настоятельно рекомендуем сделать это. Выполните перечисленные ниже действия.

1. Убедитесь, что пакет JDK установлен, а имя каталога `jdk/bin` находится в списке путей к выполняемым файлам.
2. Откройте окно для ввода команд оболочки.
3. Перейдите в каталог *jdk* (т.е. в каталог `/usr/local/jdk5.0` или `C:/jdk5.0`).
4. Создайте подкаталог `src`.

   ```
   mkdir src
   cd src
   ```

5. Выполните следующую команду:

   ```
   jar xvf ../src.jar
   ```

   (или, если вы работаете в системе Windows, `jar xvf ..\src.zip`).

Файл `src.sip` содержит исходные тексты всех общедоступных библиотек. Чтобы получить дополнительные исходные тексты (компилятора, виртуальной машины, платформенно-ориентированных методов и вспомогательных классов), посетите Web-страницу `http://www.sun.com/software/communitysource/java2`.

Документация содержится в отдельном архиве, который может иметь различные расширения (`.zip`, `.gz` и `.Z`). Ее можно скопировать с Web-страницы `http://java.sun.com/docs`. Выберите тот формат архива, который вам подходит. Если сомневаетесь, остановитесь на формате `.zip`, поскольку такой файл можно распаковать с помощью программы `jar`, входящей в пакет JDK. Если вы решили использовать программу `jar`, выполните следующие действия.

1. Убедитесь, что пакет JDK установлен правильно, а имя каталога *jdk*`/bin` находится в списке путей к выполняемым файлам.
2. Скопируйте файл с расширением `.zip`, содержащий документацию, на свой компьютер и поместите его в каталог *jdk*. Файл с документацией называется `j2sdk`*версия*`-doc.zip`, где вместо слова *версия* указан конкретный номер реализации, например 5_0.
3. Откройте окно для ввода команд.

4. Перейдите в каталог *jdk*.
5. Выполните следующую команду:

```
jar xvf j2sdkверсия-doc.zip
```

Здесь, как и ранее, *версия* означает номер реализации.

Установка примеров программ

В процессе работы вам, вероятно, понадобятся примеры программ, приведенные в книге. Вы можете найти их на Web-странице `http://www.phptr.com/corejava`. Программы хранятся в архиве `corejava.zip`, который нужно распаковать в отдельный каталог, — мы рекомендуем назвать его `CoreJavaBook`. Для распаковки можно применять любую утилиту, позволяющую работать с `zip`-файлами, например программу WinZip (*которую можно найти по адресу* `http://www.winzip.com`), *либо использовать* утилиту `jar`, входящую в состав JDK. Если вы решили использовать программу `jar`, выполните следующие действия.

1. Убедитесь, что пакет JDK установлен правильно, а каталог *jdk*`/bin` находится в списке путей к выполняемым файлам.
2. Создайте каталог `CoreJavaBook`.
3. *Скопируйте файл* `corejava.zip` *в этот каталог.*
4. Откройте окно для ввода команд.
5. Перейдите в каталог `CoreJavaBook`.
6. Выполните следующую команду:

```
jar xvf corejava.zip
```

Навигация по каталогам

Изучая язык Java, вы, вероятно, захотите просмотреть исходные тексты программ. Кроме того, вам, без сомнения, придется интенсивно работать с документацией. В табл. 2.1 показано дерево каталогов, относящихся к языку Java. Схема этих каталогов может измениться, если вы работаете в интегрированной среде разработки программ, а имя корневого каталога зависит от номера инсталлированной версии.

Таблица 2.1. Дерево каталогов JDK

Структура каталогов	Описание
`jdk`	(Имя может быть другим, например `jdk5.0`)
`bin`	Компилятор и другие инструментальные средства
`demo`	Демонстрационные программы
`docs`	*Библиотечная документация в формате HTML (после распаковки архива* `j2sdk`*версия*`-doc.zip`)
`include`	Файлы для создания платформенно-ориентированных методов (см. том 2)
`jre`	Файлы для организации среды выполнения Java-программ
`lib`	Библиотеки
`src`	Исходные тексты библиотек (после распаковки файла `src.jar`)

Для начинающих разработчиков, только приступающих к изучению языка Java, наиболее важными среди каталогов, приведенных в табл. 2.1, являются `docs` и `src`. Каталог `docs` содержит документацию о библиотеках Java в формате HTML. Их можно просматривать с помощью Web-браузера, например Netscape Navigator.

Установите в вашем браузере закладку на локальную версию файла `docs/api/index.html`. При изучении языка Java вам придется не раз обращаться к этой странице.

Каталог `src` содержит исходные тексты программ, находящихся в общедоступных библиотеках Java. Освоившись с языком, вы можете оказаться в ситуации, когда ни информация из глобальной сети, ни эта книга не помогут вам решить вашу проблему. В этом случае воспользуйтесь исходными кодами. По исходным текстам вы всегда сможете понять, что именно делает та или иная библиотечная функция. Например, если вы интересуетесь, как работает класс `System`, загляните в файл `src/java/lang/System.java`.

Выбор среды разработки программ

Если раньше вы программировали на языке Visual Basic или Visual C++, значит, вы уже знакомы со средой разработки, содержащей встроенный текстовый редактор, меню для компиляции и запуска программ, а также отладчик. Пакет JDK не имеет никаких средств, даже отдаленно напоминающих интегрированную среду разработки программ. *Все* команды выполняются в командной строке. Мы рассказываем, как инсталлировать и использовать пакет JDK, поскольку считаем, что развитая среда разработки далеко не обязательно облегчает процесс изучения языка; она может скрывать некоторые важные детали, представляющие интерес для программиста.

Интегрированные среды разработки программ не совсем удобны для написания простых программ, поскольку они работают медленно, потребляют большой объем ресурсов и, работая с ними, приходится устанавливать переменные окружения для каждой вновь создаваемой программы. Интегрированные среды полезны, если вы разрабатываете большой проект, состоящий из многих файлов. Кроме того, в состав этих сред входит отладчик, крайне необходимый при разработке серьезных программ (отладчик, работающий в режиме командной строки, поставляемый в составе JDK, крайне неудобно использовать). Мы покажем, как работать с программой Eclipse, представляющей собой свободно распространяемую среду разработки, которая сама написана на языке Java. Разумеется, если у вас уже есть среда разработки программ, например NetBeans или JBuilder, поддерживающая текущую версию языка Java, вы можете смело использовать ее при рассмотрении примеров из этой книги.

Для простых программ удобно использовать нечто среднее между работой в режиме командной строки и интегрированной средой разработки, а именно: текстовый редактор, интегрированный с пакетом JDK. На платформе Linux для этой цели можно использовать текстовый редактор Emacs. Под управлением операционной системы Windows удобна программа TextPad — прекрасный, свободно распространяемый редактор программ для среды Windows, хорошо интегрируемый со средствами Java. Тем же, кого интересуют кроссплатформенные инструменты, можно посоветовать JEdit. Используя текстовый редактор, интегрированный с JDK, можно легко и быстро создавать программы на языке Java. Большинство программ, приведенных в этой книге, разработано именно так. Поскольку компилировать и выполнять программы мож-

но, не выходя из текстового редактора, то он фактически формирует простую интегрированную среду.

Итак, у вас есть три варианта дальнейшей работы.

- Использовать набор инструментальных средств JDK вместе со своим любимым текстовым редактором. Компиляция и запуск программы на выполнение в этом случае производится из командной строки.
- Использовать интегрированную среду разработки программ, например Eclipse, либо другой свободно распространяемый или коммерческий продукт.
- *Использовать JDK и текстовый редактор, интегрированный вместе с этим* пакетом. Такую возможность предоставляют Emacs, TextPad, JEdit и многие другие подобные приложения. В этом случае компиляция и запуск на выполнение программ производятся внутри редактора.

Использование инструментов, вызываемых из командной строки

Начнем работать в самом сложном режиме: будем вызывать компилятор и запускать программы на выполнение из командной строки.

Откройте окно для ввода команд и перейдите в каталог `CoreJavaBook/v1ch2/Welcome`. (Если вы помните, каталог `CoreJavaBook` был специально создан для хранения кодов примеров из данной книги.)

Введите следующие команды:

```
javac Welcome.java
java Welcome
```

На экране должно появиться сообщение, показанное на рис.2.1.

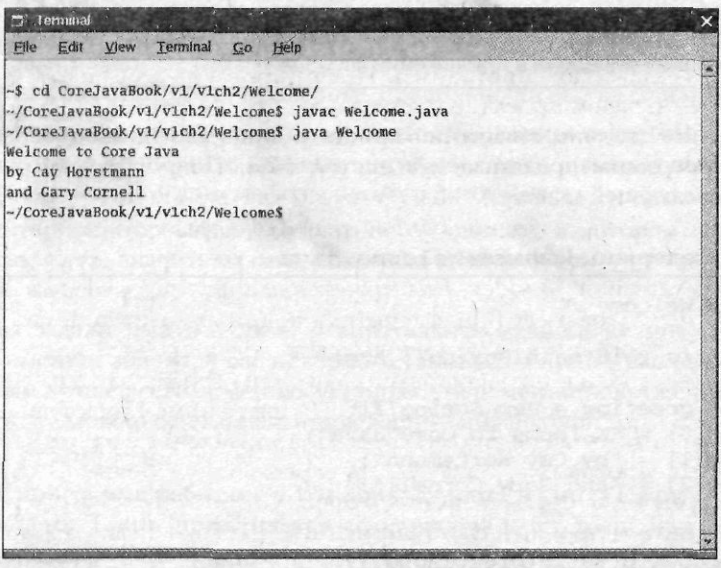


Рис. 2.1. Компиляция и выполнение программы `Welcome.java`

В системе Windows режим командной строки реализуется в окне DOS. Для того чтобы открыть это окно, надо выбрать в главном меню системы пункт Run. Если вы работаете с Windows NT/2000/XP, введите `cmd`; в противном случае вам надо задать `command`. Нажмите клавишу <Enter>, и требуемое окно появится на экране. Если вы никогда не работали в таком режиме, мы рекомендовали бы вам обратиться к специальному руководству. Подобные руководства публикуют в Web факультеты компьютерных наук многих учебных заведений. В частности, вы можете воспользоваться документом `http://www.cs.sjsu.edu/faculty/horstman/CS46A/windows/tutorial.html`.

Примите наши поздравления! Вы только что в первый раз скомпилировали и выполнили программу на языке Java.

Что же произошло? Программа `javac` — это компилятор языка Java. Она скомпилировала файл `Welcome.java` и преобразовала его в файл `Welcome.class`. Программа `java` — это интерпретатор языка Java. Он интерпретирует байтовые коды, которые компилятор записал в файл с расширением `.class`.

Если вы получили сообщение об ошибке, указывающее на приведенную ниже строку, то, вероятнее всего, используете старую версию компилятора Java. В JDK 5.0 язык был дополнен новыми возможностями, и мы воспользуемся их преимуществами.

```
for (String g : greeting)
```

Если вы все же хотите продолжать работу со старой версией Java, вам надо переписать цикл следующим образом:

```
for (int i = 0; i < greeting.length; i++)
   System.out.println(greeting[i]);
```

В данной книге мы всегда будем использовать средства из JDK 5.0. Их можно без труда преобразовать в выражения для более старых версий. Необходимая для этого информация приведена в приложении Б.

Программа `Welcome` чрезвычайно проста и лишь выводит сообщение на экран. Текст этой программы представлен в листинге 2.1. Подробнее мы рассмотрим эту программу в следующей главе.

Листинг 2.1. Содержимое файла `Welcome.java`

```
public class Welcome
{
  public static void main(String[] args)
  {
    String[] greeting = new String[3];
    greeting[0] = "Welcome to Core Java";
    greeting[1] = "by Cay Hortsmann";
    greeting[2] = "and Gary Cornell";

    for (String g : greeting)
      System.out.println(greeting[g]);
  }
}
```

Возможные ошибки

В эпоху визуальных сред разработки программ многие программисты не умеют работать в режиме командной строки. Такое неумение чревато ошибками.

Работая с командной строкой, надо учитывать следующее.

- Если вы вводите программу вручную, внимательно следите за употреблением прописных и строчных букв. В частности, в рассмотренном выше примере имя класса — `Welcome`, а не `welcome` или `WELCOME`.
- Компилятор требует указывать имя файла, в данном случае `Welcome.java`. Интерпретатору требуется *имя класса*, в данном случае это `Welcome`, без расширения `.java` или `.class`.
- Если вы получили сообщение "`Bad command or file name`" или "`javac:command not found`", проверьте, правильно ли выполнена инсталляция, в частности, верно ли указаны пути к выполняемым файлам.
- Если компилятор `javac` выдал сообщение "`cannot read: Welcome.java`", нужно проверить, имеется ли файл в соответствующем каталоге.

  При работе под управлением операционной системы Unix проверьте, правильно ли использованы прописные буквы в имени файла `Welcome.java`.

  При работе в среде Windows просматривайте содержимое каталогов с помощью команды `dir`. Некоторые текстовые редакторы (в частности, Notepad) сохраняют текст в файлах с расширением `.txt`. Если вы используете эту программу для редактирования файла `Welcome.java`, она сохранит его под именем `Welcome. java.txt`. По умолчанию в среде Windows программа Explorer скрывает расширение `.txt`, поскольку оно предполагается по умолчанию. В этом случае нужно переименовать файл, пользуясь командой `ren`, либо повторно сохранить его, указав имя в кавычках, например `"Welcome.java"`.
- Если компилятор `java` выдал сообщение об ошибке `java.lang. NoClassDefFoundError`, проверьте, правильно ли вы указали имя файла.

  Если компилятору не подходит имя `welcome` (начинающееся со строчной буквы), повторно выполните команду `java Welcome`, написав имя с прописной буквы `W`. В языке Java учитывается регистр символов.

  Если интерпретатор выдает сообщение об ошибке, в котором фигурирует имя `Welcome/java`, значит, вы случайно ввели команду `java Welcome.java`. Задайте `java Welcome`.
- Если вы задали имя `Welcome`, а виртуальная машина не может найти класс с этим именем, значит, у вас неверно установлены *пути к классам*. Для простых программ лучше всего отменить установки переменной окружения `CLASSPATH`. Сделать это можно с помощью приведенной ниже команды.

  ```
  set CLASSPATH=
  ```

  Данная программа работает в Windows, а также в Unix/Linux с оболочкой C. Если в Unix/Linux применяется оболочка Bourne/bash, надо использовать следующее выражение:

  ```
  export CLASSPATH=
  ```

  Подробно об установке путей к классам речь пойдет в главе 4.

- Если сообщение об ошибках касается новых языковых конструкций, это значит, что ваш компилятор не поддерживает JDK 5.0. Если вы по каким-то причинам не можете использовать JDK 5.0 или более поздние версии, следует изменить исходный код. Необходимая для этого информация приведена в приложении Б.
- Если в вашей программе слишком много ошибок, то все сообщения о них мелькают на экране очень быстро. Компилятор `java` выводит подобные сообщения в стандартный поток ошибок, и если информация занимает больше одного экрана, то ее желательно записать в файл.

  При работе на платформе Unix или Windows NT/2000/XP это не представляет большой сложности. Для этого можно использовать конструкцию, подобную приведенной ниже.

  ```
  javac MyProg.java 2> errors.txt
  ```

  При работе под управлением операционной системы Windows 95, находясь в режиме командной строки, невозможно перенаправить стандартный поток ошибок. В этом случае вы можете загрузить программу `errout` с Web-страницы `http://www.horstmann.com/corejava/faq.html` и выполнить команду

  ```
  errout javac Myprog.java > errors.txt
  ```

Превосходное учебное пособие приведено на Web-странице `http://java.sun.com.docs/books/tutorial/getStarted/cupojava`. В нем подробно описываются проблемы, с которыми сталкиваются начинающие программисты.

Использование интегрированной среды разработки программ

В этом разделе мы покажем, как скомпилировать программу в интегрированной среде Eclipse. Данный продукт распространяется свободно; вы можете скопировать его с Web-страницы `http://eclipse.org`. Программа Eclipse написана на языке Java, но вследствие применения нестандартных библиотек для работы с окнами переносимость ее ограничена. Тем не менее существуют версии Eclipse для операционных систем Linux, Mac OS X, Soilaris и Windows.

После запуска Eclipse активизируйте пункт меню File⇨New Project и выберите `Java Project` в диалоговом окне инструмента типа "мастер" (рис. 2.2). На данном рисунке показан продукт Eclipse 3.0M8. В вашей версии Eclipse окно может несколько отличаться.

Щелкните на кнопке Next. Укажите имя проекта `Welcome` и введите полный путь к каталогу, содержащему файл `Welcome.java` (рис. 2.3). Отмените опцию Create project in workspace и щелкните на кнопке Finish.

В результате выполненных действий вы создали новый проект. Щелкните на треугольном маркере, соответствующем окну проекта, а затем на маркере рядом с пунктом `Default package`. Дважды щелкните на `Welcome.java`. На экране отобразится окно с кодом программы, показанное на рис. 2.4.

Щелкните правой кнопкой мыши в левой панели на имени проекта (`Welcome`). В появившемся контекстном меню выберите пункт Build Project. В результате ваша программа будет скомпилирована. Если при компиляции не возникло ошибок, выберите пункт меню Run⇨Run As⇨Java Application. В нижней части окна появится панель для отображения выходных данных, на которой будут выведены результаты выполнения программы (рис. 2.5).

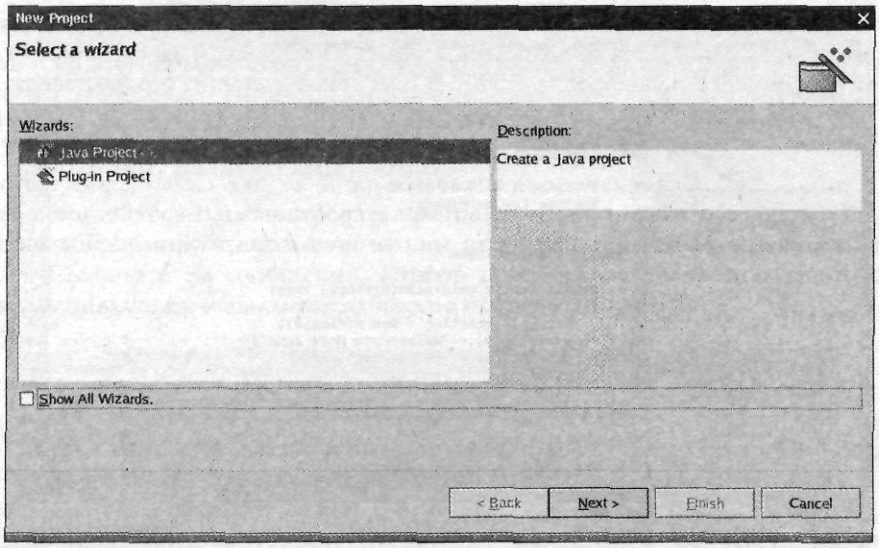


Рис. 2.2. Диалоговое окно Eclipse, предназначенное для создания проекта

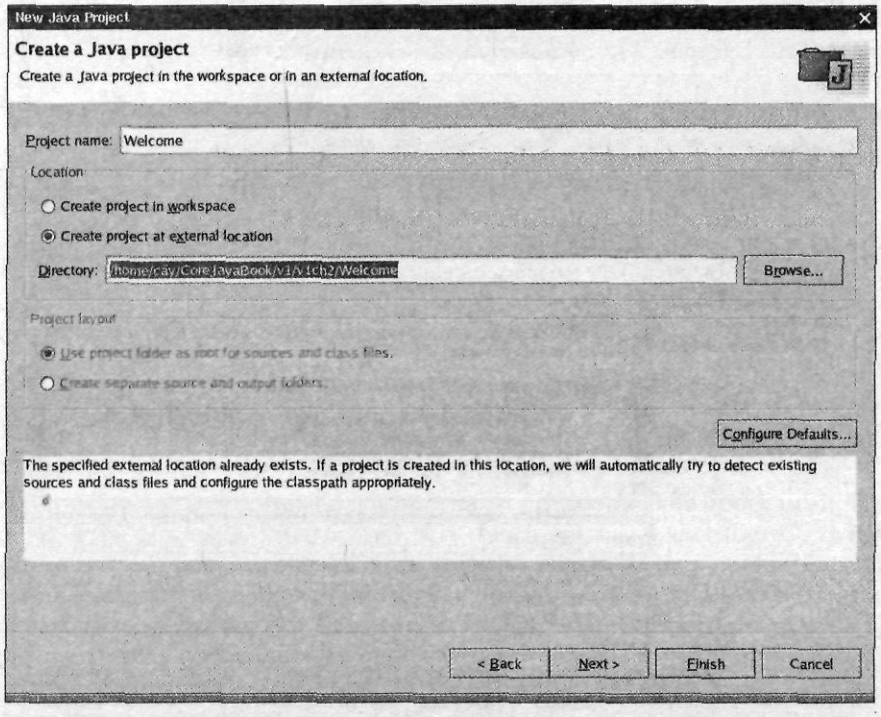


Рис. 2.3. Настройка проекта Eclipse

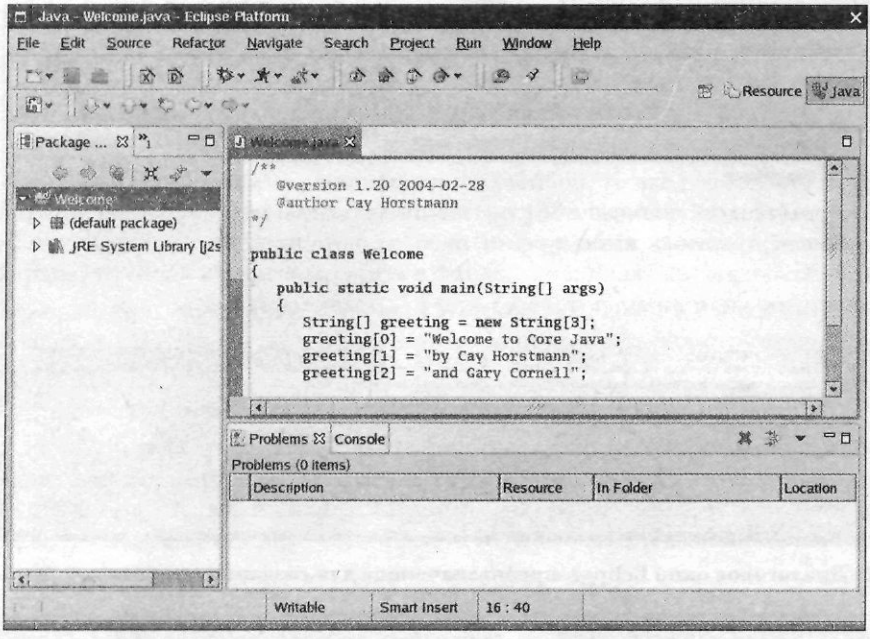


Рис. 2.4. Редактирование исходного кода при работе с Eclipse

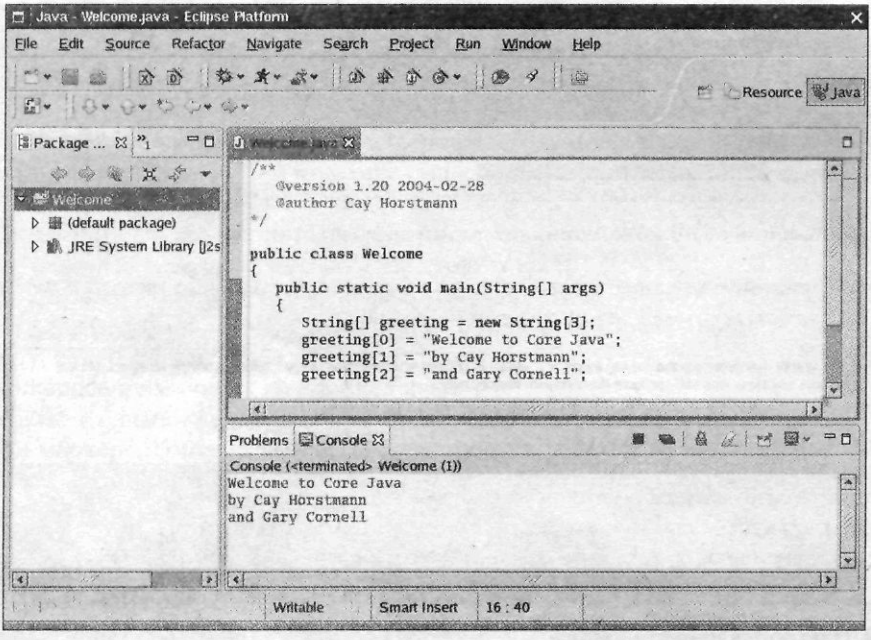


Рис. 2.5. Выполнение программы в среде Eclipse

Сообщения об ошибках компиляции

Поскольку программа состоит из нескольких строк, вряд ли в ней есть ошибка или даже опечатка. Но для того, чтобы продемонстрировать обработку ошибок, предположим, что в имени `String` вместо прописной буквы вы указали строчную:

```
public static void main(string[] args)
```

Снова попытайтесь скомпилировать программу. Вы получите сообщение об ошибке, которое информирует вас о том, что вы использовали неизвестный тип `string` (рис. 2.6). Щелкните на сообщении. Курсор перейдет на соответствующую строку в окне редактирования, и вы сможете быстро исправить ошибку.

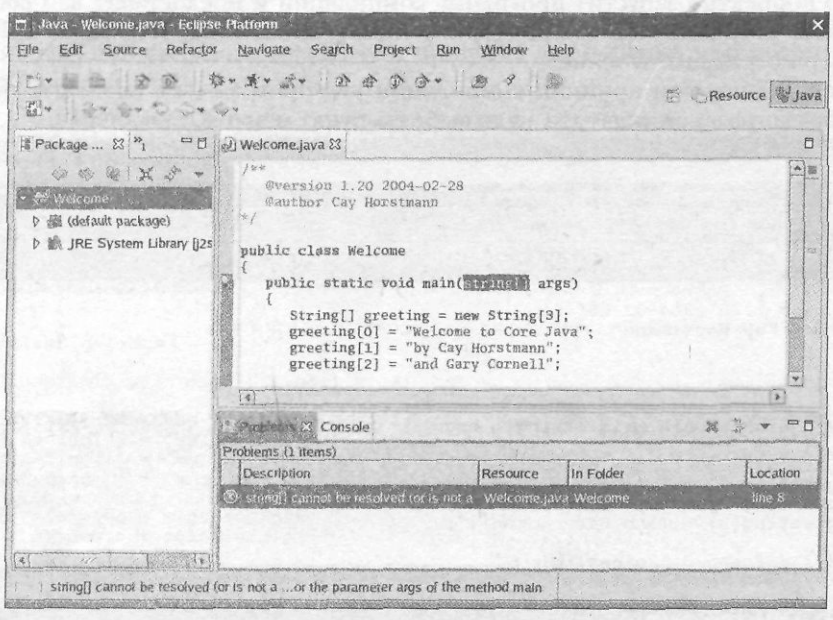


Рис. 2.6. Сообщение об ошибке, отображаемое в среде Eclipse

Таким образом, на простом примере вы получили представление о работе в среде Eclipse. Отладчик Eclipse будет рассмотрен в главе 11.

Компиляция и запуск программ из текстового редактора

Интегрированная среда разработки программ предоставляет программистам много полезных возможностей, однако у нее есть и недостатки. В частности, для простых программ, исходный текст которых содержится в одном файле, среда разработки с обилием медленно загружающихся инструментов явно избыточна. К счастью, многие текстовые редакторы позволяют запускать компилятор и интерпретатор языка Java, а также просматривать сообщения об ошибках и результаты работы программ. В этом разделе в качестве типичного примера мы рассмотрим текстовый редактор Emacs.

Emacs — это превосходный текстовый редактор, способный работать с Unix, Linux, Windows и Mac OS X. Однако многие программисты, использующие Windows, считают, что изучать его довольно сложно. Таким программистам мы можем пореко-

мендовать текстовый редактор TextPad. В отличие от Emacs, TextPad соответствует стандартам системы Windows. Информацию об этом редакторе и саму программу можно найти на Web-странице `http://www.textpad.com`. Заметим, что редактор TextPad относится к условно бесплатным программам. По истечении оговоренного срока вам придется заплатить за него. (Будучи пользователями этого продукта, мы не имеем никакого отношения к его производителям.)

Помимо упомянутых выше редакторов, мы можем предложить JEdit — свободно распространяемую программу, написанную на Java. Этот редактор можно скопировать, обратившись по адресу `http://jedit.org`. Независимо от того, выберете ли вы Emacs, TextPad, JEdit или другой редактор, основные принципы работы остаются неизменными. Закончив редактировать исходный код, можно вызывать компилятор. Редактор корректно запустит программу компиляции и предоставит вам сообщения об ошибках. Устранив ошибки, повторите компиляцию, после чего вы сможете запустить программу на выполнение.

На рис. 2.7 показан принцип компиляции Java-программы с помощью редактора Emacs. Для запуска компилятора надо выбрать пункт меню JDE⇨Compile.

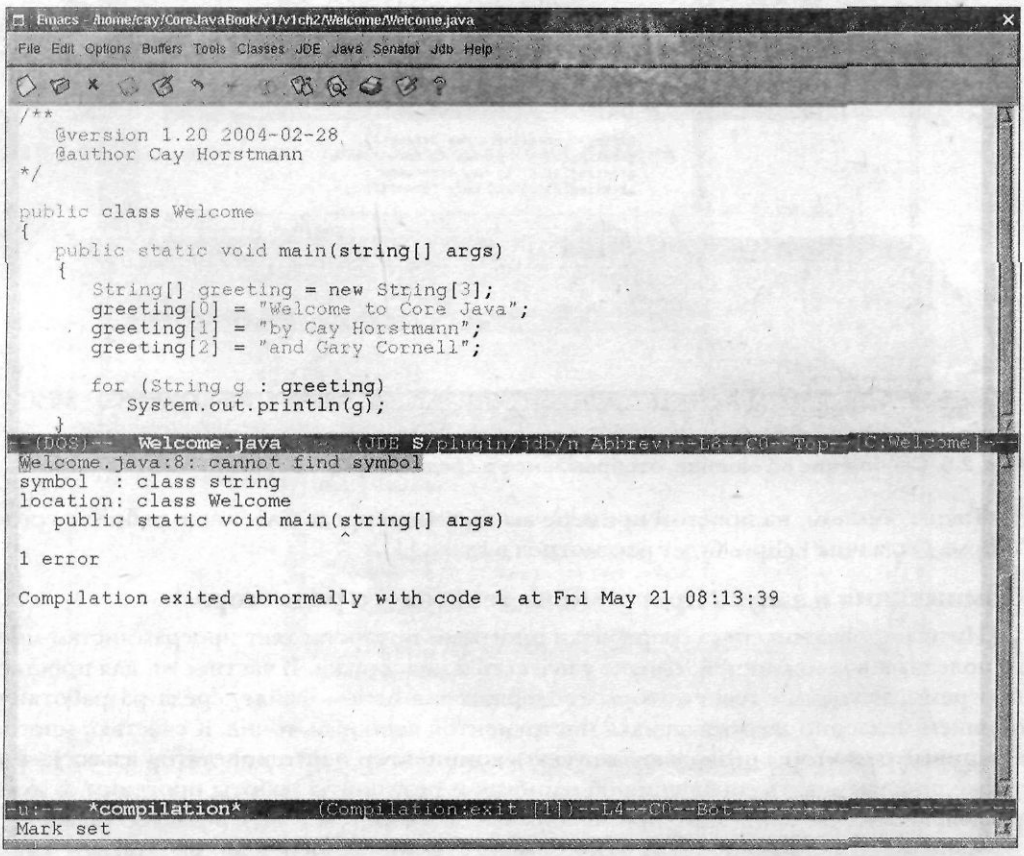


Рис. 2.7. Вызов компилятора из редактора Emacs

Текстовый редактор Emacs, созданный в рамках проекта GNU, можно загрузить с Web-страницы `http://www.gnu.org/software/emacs/`. Версию этого редактора для операционной системы Windows можно найти по адресу `http://www.gnu.org/software/emacs/windows/ntemacs.html`. Используя текстовый редактор Emacs, убедитесь, что на вашем компьютере инсталлирован продукт JDEE. Его можно скопировать с Web-страницы `http://jdee.sunsite.dk`. Для работы со средствами JDK 5.0 вам понадобится JDEE 2.4.3beta1.

Сообщения об ошибках отображаются в нижней части окна. Если вы поместите курсор на сообщение и нажмете клавишу <Enter>, курсор переместится на соответствующую строку исходного кода.

После устранения ошибок вы можете запустить программу, выбрав пункт меню JDE⇨Run App. Данные, генерируемые в процессе выполнения программы, отображаются в окне редактора (рис. 2.8).

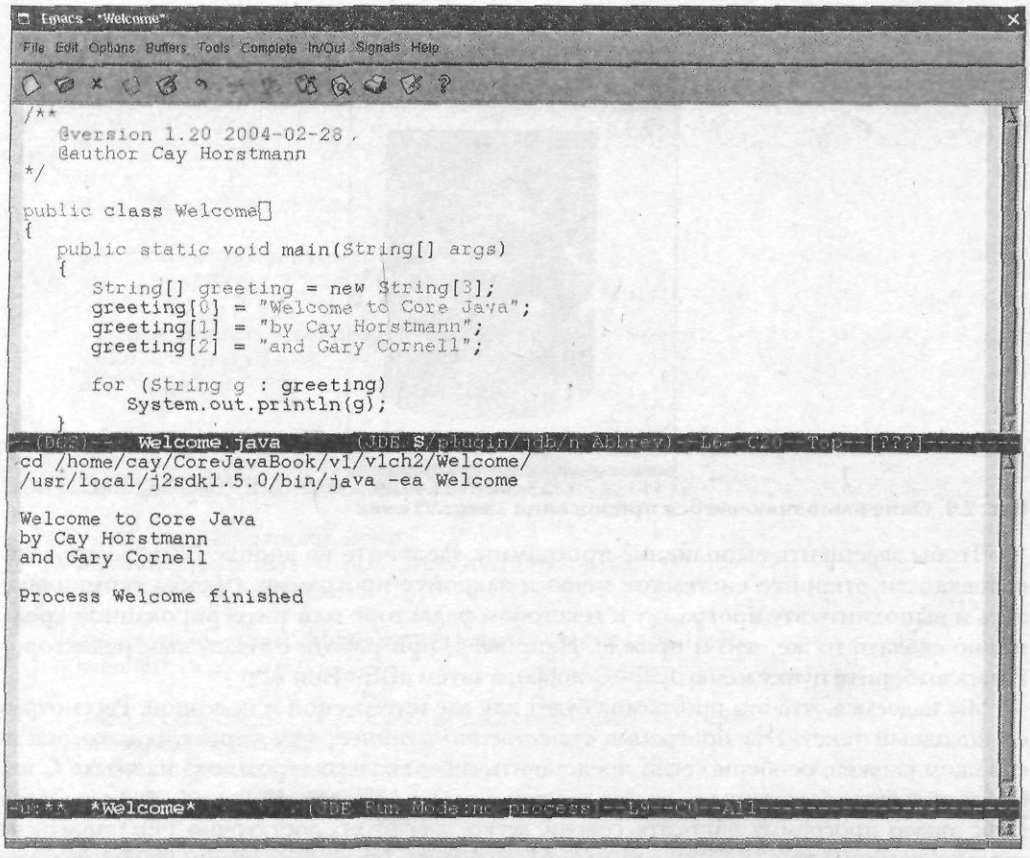


Рис. 2.8. Запуск программы из редактора Emacs

Выполнение графического приложения

Программа `Welcome` не поражает воображение. Перейдем теперь к графическому приложению. Это приложение представляет собой очень простую программу, которая выводит на экран изображение из файла. Как и прежде, скомпилируем и выполним это приложение в режиме командной строки.

1. Откройте окно для ввода команд.
2. Перейдите в каталог `CoreJavaBook/v1ch2/ImageViewer`.
3. Введите следующие команды:

```
javac ImageViewer.java
java ImageViewer
```

На экране появится новое окно приложения `ImageViewer` (рис. 2.9).

Выберите пункт меню File⇨Open и найдите файл с расширением `.gif`, который вы хотите открыть. (Мы поместили в каталог, в котором находится программа, несколько графических файлов.)

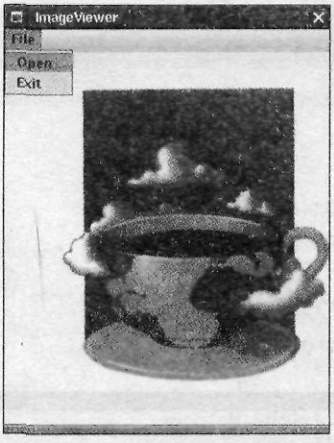


Рис. 2.9. Окно выполняющегося приложения `ImageViewer`

Чтобы завершить выполнение программы, щелкните на кнопке Close в строке заголовка или откройте системное меню и закройте программу. (Чтобы скомпилировать и выполнить эту программу в текстовом редакторе или интегрированной среде, нужно сделать то же, что и прежде. Например, при работе с текстовым редактором Emacs выберите пункт меню JDE⇨Compile, а затем JDE⇨Run App.)

Мы надеемся, что эта программа будет для вас интересной и полезной. Рассмотрим ее исходный текст. Эта программа существенно длиннее, чем первая, однако она не слишком сложна, особенно если представить себе, сколько строк кода на языке C или C++ нужно было бы написать, чтобы достичь того же эффекта. Конечно, на языке Visual Basic такую программу написать совсем легко, для этого достаточно перетащить несколько компонентов с помощью мыши и добавить несколько строк, чтобы код стал функциональным. В пакет JDK не входит инструмент для создания интерфейса, поэтому для всех элементов нужно писать соответствующий код, как показано в листинге 2.2. Написанию подобных программ посвящены главы 7–9.

Листинг 2.2. Содержимое файла `ImageViewer.java`

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;

/**
   Программа для просмотра изображений.
*/
public class ImageViewer
{
   public static void main(String[] args)
   {
      JFrame frame = new ImageViewerFrame();
      frame.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с текстовой меткой для вывода изображения.
*/
class ImageViewerFrame extends JFrame
{
   public ImageViewerFrame()
   {
      setTitle("ImageViewer");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Для вывода изображений используется метка.
      label = new JLabel();
      add(label);

      // Диалоговое окно для выбора файлов.
      chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));

      // Создание строки меню.
      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);

      JMenu menu = new JMenu("File");
      menuBar.add(menu);

      JMenuItem openItem = new JMenuItem("Open");
      menu.add(openItem);
      openItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               // Отображение диалогового окна
               // для выбора файла.
               int result = chooser.showOpenDialog(null);
```

```
                // Если файл выбран, он задается в качестве
                // пиктограммы для метки.
                if (result == JFileChooser.APPROVE_OPTION)
                {
                   String name =
                      chooser.getSelectedFile().getPath();
                   label.setIcon(new ImageIcon(name));
                }
             }
          });

      JMenuItem exitItem = new JMenuItem("Exit");
      menu.add(exitItem);
      exitItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });
   }

   private JLabel label;
   private JFileChooser chooser;
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 400;
}
```

Создание и запуск аплетов

Первые два примера, представленные в этой книге, — это *приложения*, написанные на языке Java, т.е независимые программы, аналогичные обычным платформенно-ориентированным продуктам. С другой стороны, как упоминалось в предыдущей главе, всеобщий интерес к языку Java был вызван в основном его возможностями выполнять *аплеты* под управлением Web-браузеров. В данном разделе мы рассмотрим создание и выполнение аплета в режиме командной строки. Затем мы загрузим этот аплет в специальную программу просмотра (appletviewer), входящую в состав пакета JDK. В завершение мы отобразим его в окне Web-браузера.

Перейдите в каталог `CoreJavaBook/v1ch2/WelcomeApplet`, а затем введите следующие команды:

```
javac WelcomeApplet.java
appletviewer WelcomeApplet.html
```

Внешний вид окна программы для просмотра аплетов показан на рис. 2.10.

Первая команда нам уже знакома — она вызывает компилятор языка Java. В процессе компиляции исходный текст аплета, содержащийся в файле `WelcomeApplet.java`, преобразуется в байтовый код, который помещается в файл `WelcomeApplet.class`.

Однако на этот раз мы не запускаем программу `java`. Вместо этого мы пользуемся `appletviewer`. Эта программа специально предназначена для быстрого тестирования аплетов и входит в пакет JDK. Исходными данными для `appletviewer` является файл в формате HTML. Содержимое файла `WelcomeApplet.html` показано в листинге 2.3.

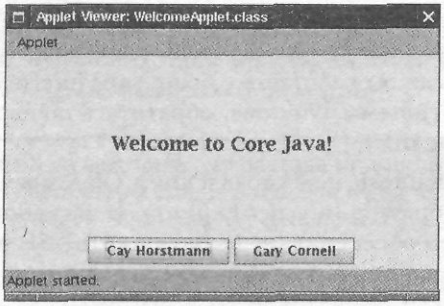


Рис. 2.10. Аплет `WelcomeApplet`, отображающийся в окне `appletviewer`

Листинг 2.3. Содержимое файла `WelcomeApplet.html`

```
<html>
   <head>
      <title>WelcomeApplet</title>
   </head>
   <body>
      <hr/>
      <p>
         This applet is from the book
         <a href="http://www.horstmann.com/corejava.html">
         Core Java</a>
         by <em>Cay Horstmann</em> and
         <em>Gary Cornell</em>,
         published by Sun Microsystems Press.
      </p>
      <applet code="WelcomeApplet.class"
      width="400" height="200">
         <param name="greeting"
         value ="Welcome to Core Java!"/>
      </applet>
      <hr/>
      <p><a href="WelcomeApplet.java">The source.</a></p>
   </body>
</html>
```

Если вы знаете язык HTML, то заметите некоторые стандартные конструкции и дескриптор `<applet>`, который указывает программе просмотра, что необходимо загрузить аплет, код которого содержится в файле `WelcomeApplet.class`. Программа `appletviewer` игнорирует все выражения HTML, за исключением дескриптора `<applet>`.

Другие дескрипторы языка HTML обрабатываются в том случае, если вы просматриваете документ с помощью браузера, поддерживающего Java 2. Однако при работе с браузерами не исключены проблемы.

- Mozilla (а также Netscape 6 и более поздние версии) поддерживают Java 2 в системах Windows, Linux и Mac OS X. Если вы хотите поэкспериментировать с аплетами, вам надо выбрать последнюю версию браузера и убедиться, что она поддерживает Java.

- Некоторые версии браузера Internet Explorer вообще не поддерживают язык Java. Другие версии поддерживают лишь очень старую виртуальную машину Java компании Microsoft. Если вы работаете с браузером Internet Explorer под управлением операционной системы Windows, обратитесь по адресу `http://java.com` и установите Java Plug-In.
- На платформе Macintosh под управлением OS X возможности языка Java уже интегрированы в браузер Internet Explorer. В частности, на момент написания книги этот браузер обеспечивает возможности J2SE версии 1.4. Система OS 9 поддерживает лишь устаревшую версию 1.1.

Убедившись, что ваш браузер поддерживает Java 2, попробуйте загрузить в него аплет. Для этого выполните следующее.

1. Запустите браузер.
2. Выберите пункт меню File⇨Open File (или откройте файл другим способом).
3. Перейдите в каталог `CoreJavaBook/v1ch2/WelcomeApplet`.

В диалоговом окне выберите файл `WelcomeApplet.html`. Браузер загрузит аплет, а также текст документа. Web-страница будет выглядеть приблизительно так, как показано на рис. 2.11.

Нетрудно заметить, что это приложение действительно интерактивно и взаимодействует с Internet. Щелкните на кнопке Cay Horstmann, и на экране отобразится страница Кея Хорстманна. Щелкните на кнопке Gary Cornell. Аплет откроет окно для ввода электронной почты, в котором уже указан адрес Гари Корнелла.

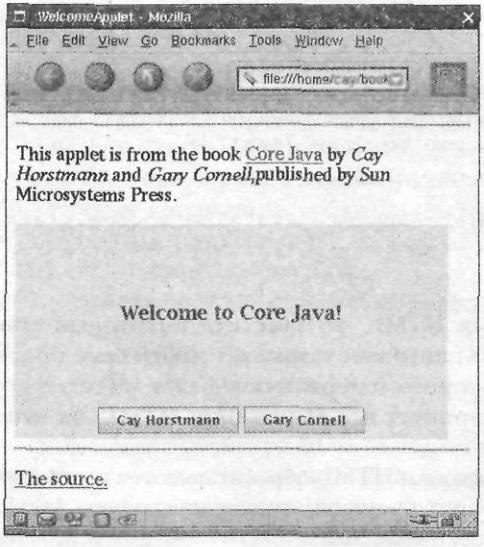


Рис. 2.11. Просмотр аплета `WelcomeApplet` с помощью браузера

Заметьте, что ни одна из этих кнопок не будет работать с программой просмотра аплетов. Эта программа не может посылать почту или отображать на экране Web-страницу, поэтому она попросту игнорирует соответствующие запросы. Про-

грамма просмотра аплетов хороша для тестирования независимых аплетов, однако для проверки реального взаимодействия аплета по сети с другими программами следует выполнить его в среде, реализуемой браузером.

Аплет можно запустить из текстового редактора или интегрированной среды разработки. При работе с редактором Emacs выберите пункт меню JDE⇨Run Applet. Работая с Eclipse, выберите пункт Run⇨Run as⇨Java Applet.

Код аплета `Welcome` показан в листинге 2.4. В данный момент его нужно лишь бегло просмотреть. К созданию аплетов мы еще вернемся в главе 10.

Итак, мы рассмотрели механизмы компиляции и запуска программ, написанных на языке Java. Теперь мы готовы перейти к главе 3, в которой приступим к изучению языка Java.

Листинг 2.4. Содержимое файла `WelcomeApplet.java`

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;

public class WelcomeApplet extends JApplet
{
   public void init()
   {
      setLayout(new BorderLayout());

      JLabel label = new JLabel(getParameter("greeting"),
         SwingConstants.CENTER);
      label.setFont(new Font("Serif", Font.BOLD, 18));
      add(label, BorderLayout.CENTER);

      JPanel panel = new JPanel();

      JButton cayButton = new JButton("Cay Horstmann");
      cayButton.addActionListener(makeURLActionListener(
         "http://www.horstmann.com"));
      panel.add(cayButton);

      JButton garyButton = new JButton("Gary Cornell");
      garyButton.addActionListener(makeURLActionListener(
         "mailto:gary@thecornells.com"));
      panel.add(garyButton);

      add(panel, BorderLayout.SOUTH);
   }

   private ActionListener makeURLActionListener(final String u)
   {
      return new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
```

```
            {
               try
               {
                  getAppletContext().showDocument(new URL(u));
               }
               catch(MalformedURLException e)
               {
                  e.printStackTrace();
               }
            }
         };
   }
}
```

Глава 3

Основные конструкции языка Java

В этой главе...

- Простая программа на языке Java
- Комментарии
- Типы данных
- Переменные
- Операторы
- Строки
- Ввод и вывод
- Поток управления
- Работа с большими числами
- Массивы

Будем считать, что вы успешно инсталлировали JDK и выполнили простые программы, приведенные в главе 2. Настало время приступать непосредственно к программированию. В этой главе вы узнаете, как в языке Java реализуются основные концепции программирования, например типы данных, ветви и циклы.

К сожалению, на языке Java нелегко написать программу с графическим интерфейсом, – для этого нужно изучить множество вопросов, связанных с окнами, полями ввода, кнопками и т.п. Поскольку описание таких технических подробностей увело бы нас далеко в сторону от нашей основной цели – анализа основных языковых конструкций, в этой главе мы рассмотрим лишь простые программы, иллюстрирующие то или иное понятие. Все эти программы используют для ввода и вывода данных консоль.

В заключение заметим, что программисты, имеющие опыт работы на языке C/C++, могут просто бегло просмотреть эту главу. Разработчикам, использующим другие языки, например Visual Basic, многие понятия также окажутся знакомыми, хотя синтаксис выражений будет существенно отличаться. Таким читателям мы советуем тщательно изучить эту главу.

Простая программа на языке Java

Рассмотрим самую простую программу, какую только можно себе представить. В процессе работы она лишь выводит сообщение в консольное окно.

```
public class FirstSample
{
   public static void main(String[] args)
   {
      System.out.println("We will not use 'Hello, World!'");
   }
}
```

Этому примеру стоит посвятить столько времени, сколько потребуется, чтобы привыкнуть к особенностям языка; и подробно рассмотреть характерные особенности Java-программ, которые будут встречаться во всех приложениях. Первое, на что надо обратить внимание, — *в языке Java учитывается регистр символов.* Если вы перепутаете их (например, введете `Main` вместо `main`), программа выполняться не будет.

Теперь просмотрим исходный код построчно. Ключевое слово `public` называется *модификатором доступа* (access modifier); такие модификаторы управляют обращением к коду из других частей программы. Более подробно мы рассмотрим этот вопрос в главе 5. Ключевое слово `class` напоминает нам, что все элементы Java-программ находятся в составе классов. Классы будут детально рассматриваться в следующей главе, а пока мы будем считать их некими "контейнерами", в которых реализована логика программы, определяющая работу приложения. Как указывалось в главе 1, классы — это "строительные блоки", из которых состоят все приложения и аплеты, написанные на языке Java.

За ключевым словом `class` следует имя класса. Правила формирования имен классов не слишком строги. Имя должно начинаться с буквы, а остальная его часть может представлять собой произвольное сочетание букв и цифр. Длина имени не ограничена. В качестве имени класса нельзя использовать зарезервированные слова языка Java (например, `public` или `class`). (Список зарезервированных слов приведен в приложении А.)

Согласно соглашениям об именовании, имя класса должно начинаться с прописной буквы (именно так сформировано имя `FirstSample`). Если имя состоит из нескольких слов, каждое из них должно начинаться с прописной буквы. (Правила, по которым в середине слова может стоять символ верхнего регистра, иногда называют "camel case", или, в соответствии с этими же правилами, `CamelCase`.)

Файл, содержащий исходный текст, должен называться так же, как и общедоступный (`public`) класс, и иметь расширение `.java`. Таким образом, код рассматриваемого здесь класса мы должны поместить в файл `FirstSample.java`. (Как и следует ожидать, регистр символов учитывается, поэтому имя `firstsample.java` не подходит.)

Если вы правильно назвали файл и не допустили ошибок в исходном тексте программы, то в результате компиляции получите файл, содержащий байтовые коды данного класса. Компилятор языка Java автоматически назовет этот файл `FirstSample.class` и сохранит его в том же каталоге, в котором содержится исходный файл. Осталось выполнить байтовые коды с помощью интерпретатора языка Java, набрав команду

```
java FirstSample
```

(Расширение `.class` не указывается!) Выполняясь, программа выведет на экран строку `"We will not use 'Hello, World'!"`.

Когда для запуска скомпилированной программы используется команда

```
java имя_класса
```

интерпретатор языка Java всегда начинает свою работу с выполнения метода `main()` указанного класса. Следовательно, чтобы программа могла выполняться, в классе *должен присутствовать* метод `main()`. Разумеется, в класс можно добавить и другие методы. (Мы покажем, как создавать такие методы, в следующей главе.)

В соответствии со спецификацией языка Java метод `main()` должен быть объявлен как `public`. (Спецификация языка Java является официальным документом. Его можно скопировать, обратившись по адресу `http://java.sun.com/docs/books/jls`.) Однако некоторые версии интерпретатора Java допускали выполнение программ, даже когда метод `main()` не имел модификатора `public`. Эта ошибка была помещена в список замеченных недостатков, представленный на сайте `http://developer.java.sun.com/developer/bugParade`, и получила номер 4252539. Однако она была помечена как "исправлению не подлежит". Разработчики компании Sun выступили с разъяснениями, что спецификация виртуальной машины языка Java не требует, чтобы метод `main()` был общедоступным (см. Web-страницу `http://java.sun.com/docs/books/vmspec`), а попытка исправить эту ошибку "может вызвать проблемы". К счастью, здравый смысл в итоге восторжествовал. Интерпретатор языка Java в пакете JDK 1.4 требует, чтобы метод `main()` был общедоступным.

Эта история не оставляет равнодушных разработчиков. С одной стороны, становится как-то неуютно от того, что инженеры, призванные гарантировать высокое качество программ, не всегда оказываются квалифицированными специалистами и позволяют себе "отмахиваться" от замеченных ошибок. С другой стороны, стоит отметить тот факт, что компания Sun разместила список ошибок и способы их исправления на Web-сайте, открыв его для всеобщего доступа. Эта информация весьма полезна для программистов. Вы даже можете проголосовать за вашу "любимую" ошибку. Ошибки, набравшие наибольшее число голосов, будут исправлены в следующих выпусках пакета JDK.

Обратите внимание на фигурные скобки в исходном тексте программы. В языке Java, так же, как и в языке C/C++, фигурные скобки используются для выделения *блоков* программы. В языке Java код любого метода должен начинаться с открывающей фигурной скобки ({) и завершаться закрывающей фигурной скобкой (}).

Расстановка фигурных скобок всегда вызвала споры. Обычно мы стараемся располагать скобки одну под другой, выравнивая их с помощью пробелов. В то же время компилятор языка Java игнорирует пробелы, поэтому, фигурные скобки можно располагать где угодно. Изучая различные операторы цикла, мы поговорим о скобках более подробно.

Пока мы не будем обращать внимание на ключевые слова `static void`, считая их просто необходимой частью программы на языке Java. В конце главы 4 мы полностью раскроем смысл этих слов. Сейчас важно помнить, что каждое приложение на языке Java должно иметь метод `main()`, заголовок которого приведен ниже.

```
public class имя_класса
{
   public static void main(String[] args)
   {
      команды
   }
}
```

Если вы программируете на языке C++, то, конечно же, знаете, что такое класс. Классы в языке Java похожи на классы в языке C/C++, однако между ними есть существенные различия. Например, в языке Java *все* методы принадлежат тому или иному классу. (Термин метод в языке Java соответствует термину функция-член в C++.) Следовательно, в языке Java должен существовать класс, которому принадлежит метод `main()`. Вероятно, вы знакомы с понятием *статических функций*-членов в языке C++. Это функции-члены, определенные внутри класса и не принадлежащие ни одному объекту. Метод `main()` в языке Java всегда является статическим. В заключение, как и в языке C/C++, ключевое слово `void` означает, что метод не возвращает никакого значения. В отличие от языка C/C++, метод `main()` не передает операционной системе код завершения. Если данный метод корректно завершает свою работу, код завершения равен 0. Чтобы изменить код завершения, надо использовать метод `System.exit()`.

Теперь обратите внимание на следующий фрагмент кода:

```
{
   System.out.println("We will not use 'Hello, World!'");
}
```

Фигурные скобки отмечают начало и конец *тела* метода, содержащего лишь одну строку кода. Как и в большинстве языков программирования, операторы языка Java можно считать предложениями. В языке Java каждый оператор должен заканчиваться точкой с запятой. В частности, символ конца строки не означает конец оператора, поэтому оператор может занимать столько строк, сколько потребуется.

В нашем случае при выполнении метода `main()` на консоль выводится одна строка текста.

В данном примере мы используем объект `System.out` и вызываем его метод `println()`. Заметьте, что метод отделяется от объекта точкой. В общем случае вызов метода имеет следующий вид:

объект.метод(параметры)

В нашем примере мы вызываем метод `println()`, передавая ему в качестве параметра текстовую строку. Метод выводит строку текста на консоль, дополняя ее символом перевода строки. В языке Java, как и в языке C/C++, строковый литерал помещается в двойные кавычки. (Далее в этой главе мы рассмотрим работу со строками подробнее.)

Методам в языке Java, как и функциям в любом другом языке программирования, может передаваться один или несколько *параметров*; метод также может вызываться без параметров. (В некоторых языках параметры принято называть *аргументами*). Даже если метод не имеет параметров, после его имени надо ставить скобки. Например, при вызове метода `println()` без параметров на экран выводится пустая строка. Такой вызов выглядит следующим образом:

```
System.out.println();
```

В объекте `System.out` есть метод `print()`, который выводит строку текста, не добавляя к ней символ перехода на новую строку. Например, выражение `System.out.print("Hello")` выводит текст "Hello" и оставляет курсор в конце строки. Следующие данные, выводимые на экран, появятся сразу за буквой "o".

Комментарии

Комментарии в языке Java, как и в большинстве языков программирования, игнорируются при выполнении программы. Таким образом, в программу можно добавлять столько комментариев, сколько потребуется, не опасаясь увеличить ее объем. В языке Java есть три способа выделения комментариев в тексте. Чаще всего используются две косые черты (//), при этом комментарий начинается сразу за символами // и продолжается до конца строки.

```
System.out.println("We will not use 'Hello, World!'");
// Остроумно, не правда ли?
```

Если нужны комментарии, состоящие из нескольких строк, можно каждую строку начинать символами //. Кроме того, для создания больших блоков комментариев можно использовать разделители /* и */, как показано в листинге 3.1.

Листинг 3.1. Содержимое файла `FirstSample.java`

```
/*
   Это первый пример программы в главе 3
   Copyright (C) 1997 Cay Horstmann and Gary Cornell
*/

public class FirstSample
{
  public static void main(String[] args)
  {
  System.out.println("We will not use 'Hello, World!'");
  }
 }
```

В заключение отметим, что в языке Java есть и третья разновидность комментариев, которую можно использовать для автоматической генерации документации. Эти комментарии начинаются символами /** и заканчиваются символами */. Более подробную информацию об этом виде комментариев и автоматической генерации документации можно найти в главе 4.

Комментарии, выделяемые символами /* и */, в языке Java не могут быть вложенными. Это значит, что фрагмент кода нельзя отключить, просто окружив его парами символов /* и */, поскольку в составе этого кода в свою очередь могут содержаться разделители /* и */.

Типы данных

Язык Java является *строго типизированным*. Это значит, что тип каждой переменной должен быть объявлен. В языке Java есть восемь *основных, или простых типов* (primitive types) данных. Четыре из них представляют целые числа, два — действительные числа с плавающей точкой, один — символы в формате Unicode и последний — логические значения.

В языке Java предусмотрен пакет для выполнения арифметических действий с произвольной точностью. Однако так называемые "большие числа" в языке Java являются *объектами*. Позднее в этой главе мы покажем, как с ними работать.

Целые числа

Целочисленные типы используются для представления как положительных, так и отрицательных чисел, не имеющих дробной части. В языке Java есть четыре целочисленных типа. Они представлены в табл. 3.1.

Таблица 3.1. Целочисленные типы в языке Java

Тип	Требуемый объем памяти (байты)	Диапазон (включительно)
`int`	4	от -2147483648 до 2147483647 (больше 2 миллиардов)
`short`	2	от -32768 до 32767
`long`	8	от -9223372036854775808 до -9223372036854775807
`byte`	1	от -128 до 127

В большинстве случаев тип `int` наиболее удобен. Если нужно задать количество жителей в самой густонаселенной стране, нет никакой необходимости прибегать к типу `long`. Типы `byte` и `short` в основном используются в специальных приложениях, например, при низкоуровневой обработке файлов или с целью экономии памяти при формировании больших массивов.

В языке Java диапазоны целочисленных типов не зависят от машины, на которой выполняется программа. Это существенно упрощает перенос программного обеспечения с одной платформы на другую. Сравните данный подход с языками C и C++, где для каждого конкретного процессора используется тип, наиболее эффективный именно на нем. В результате программа на языке C, которая отлично работает на 32-разрядном процессоре, может привести к целочисленному переполнению в 16-разрядной системе.

Длинные целые числа имеют суффикс `L` (например, `4000000000L`). Шестнадцатеричные числа имеют префикс `0x` (например, `0xCAFE`). Восьмеричные числа имеют префикс `0`. Например, `010` — это число 8. Такая запись иногда приводит к недоразумениям, поэтому мы не рекомендуем применять восьмеричные числа.

В языках C и C++ `int` означает целочисленный тип, зависящий от машины, для которой предназначена программа. На 16-разрядном процессоре, например процессоре 8086, целые числа занимают 2 байта. На 32-разрядном процессоре, например процессоре Sun SPARK, они занимают 4 байта. На процессоре Intel Pentium размер целого типа в языках C и C++ зависит от операционной системы: в DOS и Windows 3.1 целые числа занимают 2 байта. При использовании 32-разрядного режима работы в системе Windows целые числа занимают 4 байта. В языке Java размеры всех числовых типов не зависят от платформы, на которой выполняется программа.

Заметим, что в языке Java нет беззнаковых типов `unsigned`.

Числа с плавающей точкой

Типы с плавающей точкой представляют значения, имеющие дробную часть. В языке Java есть два типа для чисел с плавающей точкой, они приведены в табл. 3.2.

Таблица 3.2. Числа с плавающей точкой в языке Java

Тип	Требуемый объем памяти (байты)	Диапазон
`float`	4	Приблизительно ±3,40282347E+38F (6-7 значащих десятичных цифр)
`double`	8	*Приблизительно* ±1,7976931348623157E+308F (15 значащих десятичных цифр)

Имя `double` означает, что точность этих чисел вдвое превышает точность чисел типа `float`. (Некоторые называют их числами *с двойной точностью*.) В большинстве приложений тип `double` является наиболее удобным. Ограниченной точности чисел типа `float` во многих случаях попросту недостаточно. Семи значимых (десятичных) *цифр, возможно, хватит для того, чтобы точно выразить вашу годовую зарплату* в долларах и центах, но не зарплату президента вашей компании. Причины, по которой тип `float` все еще используется, – это скорость обработки чисел (для чисел типа `float` она выше) и экономия памяти при хранении (это важно для больших массивов действительных чисел).

Числа типа `float` имеют суффикс `F`, например `3.402F`. Числа с плавающей точкой, не имеющие суффикса `F` (например, `3.402`), всегда рассматриваются как числа типа `double`. Для их представления можно (но не обязательно) использовать суффикс `D`, например `3.402D`.

В JDK 5.0 допустимо задавать числа с плавающей точкой в шестнадцатеричном формате. Например, 0,125 – то же, что 0x1.0p-3. В шестнадцатеричной записи для указания степени вместо e используется p.

Все вычисления, производящиеся над числами с плавающей точкой, следуют стандарту IEEE 754. В частности, в языке Java есть три специальных значения с плавающей точкой:

- положительная бесконечность;
- отрицательная бесконечность;
- NaN (не число).

Они используются для обозначения переполнения и ошибок. Например, результат деления положительного числа на 0 равен положительной бесконечности. Вычисление 0/0 или извлечение квадратного корня из отрицательного числа равно NaN.

В языке Java существуют константы `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY` и `Double.NaN` (а также соответствующие константы типа `float`). Однако на практике они редко используются. В частности, для того, чтобы убедиться, что некий результат равен константе `Double.NaN`, нельзя выполнить проверку

```
if (x == Double.NAN) // Так нельзя проверить,
                     // является ли результат числом.
```

Все величины, "не являющиеся числами", считаются разными. Однако можно вызывать метод `Double.isNaN()`:

```
if (Double.isNaN(x)) // Такая проверка допустима.
```

Числа с плавающей точкой нельзя использовать в финансовых вычислениях, где ошибки округления недопустимы. Например, в результате выполнения команды `System.out.println(2.0 - 1.1)` будет выведено не 0,9, как логично ожидать, а 0,8999999999999999. Подобные ошибки связаны с внутренним двоичным представлением чисел. Подобно тому, как в десятичной системе счисления нельзя точно представить результат деления 1/3, так и в двоичной системе невозможно точно представить 1/10. Если вам надо исключить ошибки округления, следует использовать класс BigDecimal, который будет рассмотрен далее в этой главе.

Символы

Для того чтобы правильно использовать тип `char`, надо иметь представление о принципах кодировки Unicode. До появления Unicode существовало несколько различных стандартов: ASCII, ISO 8859-1, KOI-8, GB18030, BIG-5 и т.д. При этом возникали две проблемы. Во-первых, один и тот же код в разных кодировках соответствовал различным символам. Во-вторых, в языках с большим набором символов использовался код различной длины: часто употребляющиеся символы представлялись одним байтом, другие знаки – двумя, тремя и большим количеством байтов.

Для решения этих проблем была разработана кодировка Unicode. В результате исследований, направленных на унификацию кодов символов, выяснилось, что двухбайтового кода более чем достаточно для представления всех символов, использующихся во всех языках; при этом оставался достаточный резерв для любых мыслимых расширений. В 1991 г. была выпущена спецификация Unicode 1.0, в которой было использовано меньше половины из возможных 65536 кодов. В Java изначально были приняты 16-битовые символы Unicode, что стало еще одним преимуществом перед другими языками.

Однако впоследствии случилось непредвиденное: количество символов превысило допустимые 65536. Причиной тому стали чрезвычайно большие наборы иероглифов китайского, японского и корейского языков. Поэтому в настоящее время 16-битового типа `char` недостаточно для описания всех символов Unicode.

Чтобы понять, как эта проблема решается в Java, начиная с JDK 5.0, надо ввести несколько терминов. Назовем *кодовой точкой* (code point) значение, связанное с символом в схеме кодирования. Согласно стандарту Unicode, кодовые точки записываются в шестнадцатеричном формате и предваряются символами U+. Например, для буквы A кодовая точка равна U+0041. В Unicode кодовые точки объединяются в 17 *кодовых плоскостей* (code plane). Первая кодовая плоскость, называемая *основной многоязыковой плоскостью* (basic multilingual plane), состоит из "классических" символов Unicode с кодовыми точками от U+0000 до U+FFFF. Шестнадцать дополнительных плоскостей с кодовыми точками от U+10000 до U+10FFFF содержат *дополнительные символы* (supplementary character).

Кодировка UTF-16 – это способ представления всех кодов Unicode последовательностью переменной длины. Символы из основной многоязыковой плоскости представляются 16-битовыми значениями, называемыми *кодовыми единицами* (code unit). Дополнительные символы обозначаются последовательными парами кодовых единиц. Каждое из значений пары попадает на используемую 2048-байтовую область основной многоязыковой плоскости, называемой *областью подстановки* (surrogates area); от U+D800 до U+DBFF для первой кодовой единицы и от U+DC00 до U+DFFF для второй кодовой единицы. Такой подход позволяет сразу определить, соответствует ли значение коду конкретного символа или является ли частью кода дополнительного символа. Например, математическому коду символов, обозначающему множество целых чисел, соответствует кодовая точка U+1D56B и две кодовых единицы, U+D835 и U+DD6B (описание алгоритма кодирования можно найти по адресу `http://en.wikipedia.org/wiki/UTF-16`).

В Java тип char описывает кодовую единицу UTF-16.

Начинающим программистам мы советуем использовать коды UTF-16 лишь в случае крайней необходимости. Если это возможно, используйте строки.

В некоторых случаях применение типа char вполне оправдано. Речь идет о работе с символьными константами. Например, символьной константой является 'A', которой соответствует значение 65. Не следует путать символ 'A' со строкой "A", состоящей из одного символа. Кодовые единицы Unicode можно выражать в виде шестнадцатеричных чисел в диапазоне от \u0000 до \uFFFF. Например, значение \u2122 соответствует символу торговой марки (™), а \u03C0 — греческой букве π.

Кроме префикса \u, который предваряет кодовую единицу Unicode, существует также несколько специальных символьных последовательностей, показанных в табл. 3.3. Эти последовательности можно применять в составе символьных констант или строк, например '\u2122' или "Hello\n". Последовательности, начинающиеся с \u (и никакие другие), можно даже указывать за пределами символьных констант или строк. Приведенный ниже пример корректен, так как последовательности \u005B и \u005D соответствуют символам [и].

```
public static void main(String\u005B\u005D args)
```

Таблица 3.3. Специальные символы

Специальный символ	Описание	Значение Unicode
\b	Возврат на одну позицию	\u0008
\t	Табуляция	\u0009
\n	Переход на новую строку	\u000a
\r	Возврат каретки	\u000d
\"	Двойная кавычка	\u0022
\'	Одинарная кавычка	\u0027
\\	Обратная косая черта	\u005c

Теоретически в приложении или аплете на языке Java можно использовать любой символ в формате Unicode, однако будет ли он отображаться на экране дисплея, зависит от вашего браузера (для аплетов) и от операционной системы.

Логические значения

Для типа boolean предусмотрены два значения: false и true. Они соответствуют результатам вычисления логических выражений. Преобразование логических переменных в целочисленные и наоборот невозможно.

В языке C++ вместо логических значений можно использовать числа и даже указатели. Значение 0 эквивалентно логическому значению false, а ненулевые величины — значению true. В языке Java представлять логические значения посредством других типов невозможно. Следовательно, программист на языке Java защищен от недоразумений, подобных следующему:

```
if (x = 0) // Вместо проверки x==0 выполнили присваивание!
```

В языке C++ эта строка компилируется и выполняется проверка, причем выражение всегда равно false. В языке Java наличие такой строки приведет к ошибке на этапе *компиляции, поскольку целочисленное выражение* x = 0 *нельзя преобразовать в* логическое значение.

Переменные

В языке Java каждая переменная имеет *тип*. При объявлении переменной сначала указывается ее тип, а затем ее имя. Ниже приведено несколько примеров объявления переменных.

```
double salary;
int vacationDays;
long earthPopulation;
char yesChar;
boolean done;
```

Обратите внимание на точку с запятой в конце каждого выражения. Она необходима, поскольку объявление в языке Java считается оператором.

Имя переменной должно начинаться с буквы и представлять собой сочетание букв и цифр. Термины "буквы" и "цифры" в Java имеют более широкое значение, чем в большинстве других языков программирования. Буквами считаются символы `'A'-'Z'`, `'a'-'z'`, `'_'` и *любой другой* символ в кодировке Unicode, соответствующий букве. Например, немецкие пользователи в именах переменных могут использовать символ `'ä'`, а греческие пользователи могут воспользоваться буквой π. Аналогично цифрами считаются как обычные десятичные цифры, `'0'-'9'`, так и любые символы в кодировке Unicode, использующиеся для обозначения цифры в каком-либо языке. Символы наподобие `'+'` или `'©'`, а также пробел нельзя использовать в именах переменных. Все символы в имени переменной важны, причем *регистр также учитывается*. Длина имени переменной не ограничена.

Если вы действительно хотите знать, какие символы в формате Unicode считаются "буквами" в языке Java, воспользуйтесь методами `isJavaIdentifierStart()` и `isJavaIdentifierPart()` класса `Character`.

В качестве имен переменных нельзя использовать зарезервированные слова. (Список зарезервированных слов приведен в приложении А.)

В одной строке программы можно размещать несколько объявлений, например:

```
int i,j; // Обе переменные — целочисленные.
```

Однако мы не рекомендуем следовать такому стилю. Если объявить каждую переменную в отдельной строке, читать программу станет гораздо легче.

Как уже было сказано, в языке Java различаются прописные и строчные буквы, например имена `hireday` и `hireDay` считаются разными. Иногда для переменной трудно подобрать подходящее имя. Многие программисты в этих случаях дают переменной имя, совпадающее с именем типа, но отличающееся регистром символов. Например:

```
Box box; // -- Box — это тип, а box — имя переменной.
```

На наш взгляд, намного лучше использовать в имени переменной префикс `a`:

```
Box aBox;
```

Инициализация переменных

После объявления переменной ее нужно инициализировать с помощью оператора присваивания, поскольку использовать переменную, которой не присвоено никакого значения, невозможно. Например, приведенный ниже фрагмент кода будет признан ошибочным уже на этапе компиляции.

```
int vacationDays;
System.out.println(vacationDays);  // Ошибка!
                                   // Переменная не инициализирована.
```

Для присвоения ранее объявленной переменной какого-либо значения нужно указать слева ее имя, поставить знак равенства (=), а справа записать некоторое выражение на языке Java, задающее требуемое значение.

```
int vacationDays;
vacationDays = 12;
```

При желании вы можете одновременно объявить и инициализировать переменную. Например:

```
int vacationDays = 12;
```

В языке Java объявление переменной можно размещать в любом месте кода; например, приведенный ниже фрагмент вполне допустим.

```
double salary = 65000.0;
System.out.println(salary);
int vacationDays = 12; // Здесь можно объявить переменную.
```

При создании программ на Java рекомендуется объявлять переменную как можно ближе к той точке кода, где она будет использована.

В языках C и C++ различаются *объявление* и *определение* переменной. Ниже приведен пример определения переменной.

```
int i = 10;
```

Объявление переменной выглядит следующим образом:

```
extern int i;
```

В языке Java объявление и определение переменных не различаются.

Константы

В языке Java для обозначения констант используется ключевое слово `final`, например:

```
public class Constants
{
    public static void main(String[] args)
    {
        final double CM_PER_INCH = 2.54;
        double paperWidth = 8.5;
        double PaperHeight = 11;
        System.out.println("Размер страницы в сантиметрах: "
           + paperWidth * CM_PER_INCH + "на"
           + paperheight * CM_PER_INCH);
    }
}
```

Ключевое слово final означает, что присвоить какое-либо значение данной переменной можно лишь один раз и изменять его нельзя. Использовать в именах констант только прописные буквы необязательно.

В языке Java часто возникает необходимость в константах, доступных нескольким методам внутри одного класса. Обычно они называются *константами класса* (class constant). Константы класса объявляются с помощью ключевых слов static final. Ниже приведен пример использования константы класса.

```
public class Constants2
{
    public static void main(String[] args)
    {
        double paperWidth = 8.5;
        double PaperHeight = 11;
        System.out.println("Размер страницы в сантиметрах: "
           + paperWidth * CM_PER_INCH + "на"
           + paperHeight * CM_PER_INCH);
    }


    public static final double CM_PER_INCH = 2.54;
}
```

Константа класса задается *вне* метода main(), поэтому ее можно использовать в других методах того же класса. Более того, если (как в данном примере) константа объявлена как public, методы из других классов также могут получить к ней доступ. В нашем примере это можно сделать с помощью выражения Constants2.CM_PER_INCH.

В языке Java слово const является зарезервированным, однако сейчас оно уже не употребляется. Для объявления констант следует использовать ключевое слово final.

Операторы

Для обозначения операций сложения, вычитания, умножения и деления в языке Java используются обычные арифметические операторы + - * /. Оператор / обозначает целочисленное деление, если оба его аргумента являются целыми числами. В противном случае этот оператор обозначает деление чисел с плавающей точкой. Остаток от деления целых чисел обозначается символом %. Например, 15/2 равно 7; 15%2 равно 1; а 15.0/2 —7.5.

Заметим, что целочисленное деление на 0 генерирует исключение, в то время как результатом деления на 0 чисел с плавающей точкой является бесконечность или NaN.

В языке Java предусмотрена сокращенная запись бинарных арифметических операторов (т.е. операторов, предполагающих два операнда). Например, выражение

```
x += 4;
```

эквивалентно

```
x = x + 4;
```

(В сокращенной записи символ арифметической операции, например * или %, размещается перед знаком равенства, например *= или %=.)

Одной из заявленных целей языка Java является машинная независимость. Вычисления должны приводить к одинаковому результату, независимо от того, какая виртуальная машина их выполняет. Для арифметических вычислений над числами с плавающей точкой это неожиданно оказалось трудной задачей. Тип `double` использует для хранения числовых значений 64 бита, однако некоторые процессоры применяют 80-разрядные регистры с плавающей точкой. Эти регистры обеспечивают дополнительную точность на промежуточных этапах вычисления. Рассмотрим в качестве примера следующее выражение:

```
double w = x * y / z;
```

Многие процессоры компании Intel вычисляют выражение `x * y` и сохраняют этот промежуточный результат в 80-разрядном регистре, затем делят его на значение переменной `z` и в самом конце округляют ответ до 64 бит. Так можно повысить точность вычислений, избежав переполнения. Однако этот результат может оказаться *иным*, если в процессе всех вычислений используется 64-разрядный процессор. По этой причине в первоначальном описании виртуальной машины Java указывалось, что все промежуточные вычисления должны округляться. Это вызвало протест многих специалистов. Округление не только может привести к переполнению. Вычисления при этом происходят *медленнее*, поскольку операции округления занимают определенное время. В результате разработчики языка Java изменили свое мнение, стремясь разрешить конфликт между оптимальной производительностью и воспроизводимостью результатов. По умолчанию в виртуальной машине в промежуточных вычислениях может использоваться повышенная точность. Однако методы, помеченные ключевым словом `strictfp`, должны использовать операции над числами с плавающей точкой, гарантирующие воспроизводимость результатов. Например, метод `main()` можно записать так, как показано ниже.

```
public static strictfp void main(String[] args)
```

В этом случае все команды внутри метода `main()` будут выполнять точные операции над числами с плавающей точкой.

Детали выполнения этих операций тесно связаны с особенностями работы процессоров Intel. По умолчанию промежуточные результаты могут использовать расширенный показатель степени, но не расширенную мантиссу. (Микросхемы компании Intel поддерживают округление мантиссы без потери производительности.) Следовательно, единственное различие между вычислениями по умолчанию и точными вычислениями состоит в возможности переполнения.

Если сказанное кажется вам слишком сложным, не волнуйтесь. Переполнение при вычислениях чисел с плавающей точкой в большинстве случаев не возникает. В примерах, рассматриваемых в этой книге, ключевое слово `strictfp` использоваться не будет.

Операторы инкрементирования и декрементирования

Программисты, конечно, знают, что одной из наиболее распространенных операций с числовыми переменными является добавление или вычитание единицы. В языке Java, как и в языках C и C++, есть операторы инкрементирования и декрементирования: в результате вычисления выражения `x++` к текущему значению переменной `x` прибавляется единица, а `x--` уменьшает `x` на единицу. Например, в результате обработки следующего фрагмента значение переменной `n` становится равным 13.

```
int n = 12;
n++;
```

Поскольку эти операторы ++ и -- изменяют значение переменной, их нельзя применять к самим числам. Например, выражение 4++ является недопустимым.

Существуют два вида операторов инкрементирования и декрементирования. Выше показана постфиксная форма, в которой символы операции размещаются после операнда. Есть и префиксная форма — ++n. Оба этих оператора изменяют значение переменной на единицу. Разница между ними проявляется, только когда эти операторы присутствуют в выражениях. Префиксная форма сначала изменяет значение переменной и использует новое значение для дальнейших вычислений, а постфиксная форма использует старое значение этой переменной и лишь после этого изменяет его.

```
int m = 7;
int n = 7;
int a = 2 * ++m; // Теперь значение a равно 16, а m равно 8.
int b = 2 * n++; // Теперь значение b равно 14, а n равно 8.
```

Мы не рекомендуем использовать операторы инкрементирования и декрементирования внутри выражений, поскольку это зачастую запутывает код и приводит к досадным ошибкам.

(Поскольку именно оператор ++ дал имя языку C++, это послужило поводом к первой шутке о нем. Недоброжелатели отмечают, что даже имя этого языка содержит в себе ошибку: "Язык следовало бы назвать ++C, потому что мы хотим использовать его только после улучшения".)

Операторы отношения и логические операторы

Язык Java содержит полный комплект операторов отношения. Чтобы проверить равенство, следует использовать символы ==. Например, значение выражения `3 == 7` равно `false`.

Для проверки неравенства используются символы `!=`. Так, значение выражения `3 != 7` равно `true`.

Кроме того, в языке Java есть обычные операторы < (меньше), > (больше), <= (меньше или равно) и => (больше или равно).

В языке Java, как и в C++, используются символы && для обозначения логического оператора "и", а также символы || для обозначения логического оператора "или". Как обычно, знак восклицания означает логический оператор отрицания. Операторы && и || задают вычисление по сокращенной схеме, согласно которой, если первый элемент определяет значение всего выражения, то остальные элементы не вычисляются. Рассмотрим два выражения, объединенных оператором &&.

выражение_1 && *выражение_2*

Если первое выражение ложно, то вся конструкция не может быть истинной. Поэтому второе выражение вычислять нет смысла. Например, в приведенном ниже выражении вторая часть не вычисляется, если значение переменной x равно нулю.

```
x != 0 && 1/x > x+y // Не делить на 0.
```

Таким образом, деление на нуль не происходит.

Аналогично, значение *выражение_1* || *выражение_2* истинно, если истинным является значение первого выражения. В этом случае вычислять второе выражение нет необходимости.

В языке Java есть также тернарный оператор ?:, который иногда оказывается полезным.

```
условие ? выражение_1 : выражение_2
```

Если условие истинно, то вычисляется первое выражение, а если ложно — второе выражение. Например, `x < y ? x : y` возвращает меньшее из чисел x и y.

Побитовые операции

Работая с любыми целочисленными типами, можно применять операторы, непосредственно обрабатывающие биты, из которых состоят целые числа. Это значит, что для определения состояния отдельных битов числа можно использовать маски. В языке Java есть следующие побитовые операторы: `&` ("и"), `|` ("или"), `^`("исключающее или"), `~`("не"). Например, если n — это целое число, то приведенное ниже выражение равно единице только в том случае, если четвертый бит в двоичном представлении числа равен единице.

```
int fourthBitFromRight = (n & 8) / 8;
```

Используя оператор `&` в сочетании с соответствующей степенью двойки, можно замаскировать все биты, кроме одного.

В применении к логическим переменным операторы `&` и `|` дают логические значения. Эти операторы аналогичны операторам `&&` и `||`, за исключением того, что вычисление производится по полной схеме, т.е. обрабатываются все элементы выражения.

В языке Java есть также операторы >> и <<, сдвигающие битовое представление числа вправо или влево. Эти операторы часто оказываются удобными, если нужно построить битовое представление на основе маски:

```
int fourthBitFromRight = (n & (1 << 3)) >> 3;
```

В языке есть даже оператор >>>, заполняющий старшие разряды нулями, в то время как оператор >> восстанавливает в старших разрядах знаковый бит. Оператора <<< в языке Java нет.

Значение, присутствующее в правой части операторов побитового сдвига, сокращается по модулю 32 (если левая часть является числом типа `long`, правая часть сокращается по модулю 64). Например, значение выражения `1<<35` эквивалентно выражению `1<<3`, или 8.

В языках C/C++ не определено, какой сдвиг выполняет оператор >>: арифметический (при котором знаковый бит восстанавливается) или логический (при котором старшие разряды заполняются нулями). Разработчики средств реализации языка могут выбрать тот вариант, который покажется им более эффективным. Это значит, что результат выполнения операции сдвига вправо в языке C/C++ определен лишь для неотрицательных чисел. В языке Java данная проблема разрешена путем ввода оператора >>>.

Математические функции и константы

Класс Math содержит набор математических функций, которые часто оказываются необходимыми при решении практических задач.

Чтобы извлечь квадратный корень из числа, применяют метод sqrt().

```
double x = 4;
double y = Math.sqrt(x);
System.out.println(y); // Выводит число 2.0.
```

Между методами println() и sqrt() есть небольшая разница. Метод println() принадлежит объекту System.out. (Напомним, что out — это объект, определенный в классе System и представляющий стандартное устройство вывода.) Метод sqrt() принадлежит классу Math, а не объекту. Такие методы называются *статическими*. Они будут рассматриваться в главе 4.

В языке Java нет оператора возведения в степень: для этого нужно использовать метод pow() класса Math. В результате выполнения следующей строки кода переменной y присваивается значение переменной x, возведенное в степень a (x^a).

```
double y = Math.pow(x,a);
```

Оба параметра метода pow(), а также возвращаемое им значение имеют тип double.

Класс Math содержит методы для вычисления обычных тригонометрических функций:

```
Math.sin()
Math.cos()
Math.tan()
Math.atan()
Math.atan2()
```

Кроме того, в него включены экспоненциальная и обратная к ней логарифмическая функции (натуральный логарифм):

```
Math.exp
Math.log
```

В данном классе также определены две константы — приближенное представление чисел π и *e*.

```
Math.PI
Math.E
```

Начиная с JDK 5.0, при вызове методов для математических вычислений класс Math можно не указывать, включив вместо этого в начало файла с исходным кодом следующее выражение:

```
import static java.lang.Math.*;
```

Например, при компиляции приведенной ниже строки кода ошибка не возникает.

```
System.out.println("The square root of \u03C0 is " + sqrt(PI));
```

Подробно вопросы статического импортирования мы рассмотрим в главе 4.

Для повышения своей производительности функции в классе `Math` используют программы из аппаратного модуля, предназначенного для вычислений с плавающей точкой. Если для вас не очень существенна скорость работы, а важнее получить предсказуемые результаты, используйте класс `StrictMath`. Он реализует алгоритмы из библиотеки `fdlibm`, гарантирующей идентичность результатов на всех платформах. Исходные тексты программ, реализующих эти алгоритмы, можно найти на Web-странице `http://www.netlib.org/fdlibm/index.html`. (Поскольку в библиотеке `fdlibm` каждая функция определена неоднократно, в классе `StrictMath` в соответствии с IEEE 754 имена функций начинаются с буквы "e".)

Преобразование числовых типов

Часто возникает необходимость преобразовать один числовой тип в другой. На рис. 3.1 показаны допустимые преобразования.

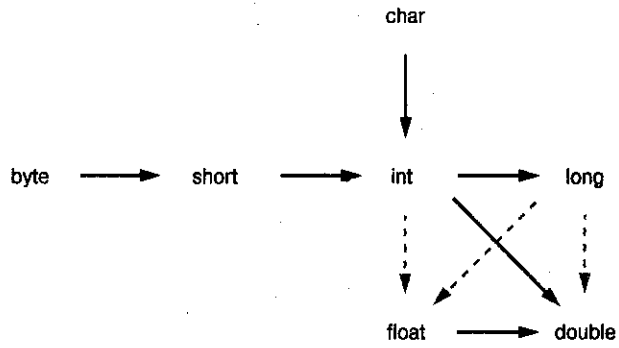


Рис. 3.1. Допустимые преобразования числовых типов

Шесть сплошных линий со стрелками обозначают преобразования, которые выполняются без потери информации. Три штриховые линии, также со стрелками, означают преобразования, при которых может произойти потеря точности. Например, количество цифр в длинном целом числе 123456789 превышает количество цифр, которое может быть представлено типом `float`. Число, преобразованное в тип `float`, имеет тот же порядок, но несколько меньшую точность.

```
int n = 123456789;
float f = n; // Содержимое f равно 1.234567892E8.
```

Если два значения объединяются бинарным оператором (например, `n+f`, где `n` — целое число, а `f` — число с плавающей точкой), то перед выполнением операции оба операнда преобразовываются в числа одинакового типа.

- Если хотя бы один из операндов имеет тип `double`, то второй тоже преобразовывается в число типа `double`.
- В противном случае, если хотя бы один из операндов имеет тип `float`, то второй тоже преобразовывается в тип `float`.
- В противном случае, если хотя бы один из операндов имеет тип `long`, то второй тоже преобразовывается в число типа `long`.
- В противном случае оба операнда преобразовываются в числа типа `int`.

Приведение числовых типов

Как уже было сказано, при необходимости значения типа `int` автоматически преобразовываются в значения типа `double`. С другой стороны, в ряде ситуаций число типа `double` должно рассматриваться как целое. Преобразования чисел в языке Java возможны, однако, разумеется, при этом может происходить потеря информации. Такие преобразования называются *приведением типов* (cast). Синтаксически приведение типа задается парой скобок, внутри которых указывается желательный тип, а затем имя переменной. Например:

```
double x = 9.997;
int nx = (int)x;
```

Теперь в результате приведения значения с плавающей точкой к целому типу переменная nx равна 9, поскольку при этом дробная часть числа отбрасывается.

Если нужно *округлить* число с плавающей точкой до *ближайшего* целого числа (что во многих случаях является намного более полезным), используется метод `Math.round()`.

```
double x = 9.997;
int nx = (int)Math.round(x);
```

Теперь переменная nx равна 10. При вызове метода `round()` по-прежнему нужно выполнять приведение `(int)`, поскольку возвращаемое им значение имеет тип `long`.

При попытке приведения типов результат может выйти за пределы допустимого диапазона. В этом случае произойдет усечение. Например, при вычислении выражения `(byte)300` будет получено значение 44.

Приведение логических значений к целым и наоборот невозможно. Это предотвращает появление ошибок. В редких случаях, когда действительно необходимо представить логическое значение в виде целого, можно использовать условное выражение `b ? 1 : 0`.

Иерархия операторов

В табл. 3.4 приведена информация о приоритете операторов. Если скобки не используются, сначала выполняются более приоритетные операции. Операторы, находящиеся на одном уровне иерархии, выполняются слева направо, за исключением операторов, имеющих правую ассоциативность, как показано в таблице. Например, поскольку оператор `&&` приоритетнее `||`, выражение `a && b || c` эквивалентно `(a && b) || c`. Так как оператор += ассоциируется справа налево, выражение `a += b += c` означает `a += (b += c)`. В данном случае значение `b += c` (значение b после прибавления к нему значения c) присваивается переменной a.

В отличие от языков C и C++, язык Java не имеет оператора "запятая". Однако в операторе `for` в качестве первого и третьего операторов можно использовать список выражений, разделенных запятыми.

Таблица 3.4. Приоритет операторов

Операторы	Ассоциативность
[] . () (вызов метода)	Слева направо
! ~ ++ -- +(унарный) -(унарный) ()(приведение) new	Справа налево
* / %	Слева направо
+ -	Слева направо
<< >> >>>	Слева направо
< <= > >= instanceof	Слева направо
== !=	Слева направо
&	Слева направо
^	Слева направо
|	Слева направо
&&	Слева направо
||	Слева направо
?:	Справа налево
= += -= *= /= %= |= ^= <<= >>= >>>=	Справа налево

Нумерованные типы

В некоторых случаях переменной должны присваиваться лишь значения из ограниченного набора. Предположим, например, что вы продаете пиццу четырех размеров: малого, среднего, большого и очень большого. Конечно, вы можете представить размеры целыми числами, например 1, 2, 3 и 4, или буквами `S`, `M`, `L` и `X`. Однако такой подход чреват ошибками. В процессе составления программы можно присвоить переменой недопустимое значение, например 0 или `m`.

В JDK 5.0 появилась возможность использовать в подобных ситуациях *нумерованный тип*. Например, вы можете включить в программу приведенное ниже выражение.

```
enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

После этого можно определять переменные данного типа.

```
Size s = Size.MEDIUM;
```

Переменная типа `Size` может содержать только предопределенные значения. Допустимо также значение `null`, указывающее на то, что данной переменной не присвоена никакая величина.

Подробно нумерованные типы будут рассмотрены в главе 5.

Строки

Строка Java — это последовательность символов Unicode. Например, строка `"Java\u2122"` состоит из пяти символов: `J`, `a`, `v`, `a` и ™. В языке Java нет встроенного типа для строк. Вместо этого стандартная библиотека языка содержит класс `String`. Каждая строка, помещенная в кавычки, представляет собой экземпляр класса `String`.

```
String e = ""; // Пустая строка.
String greeting = "Hello";
```

Кодовые точки и кодовые единицы

В языке Java строки реализованы как последовательности значений типа char. Как было сказано ранее, тип char позволяет задавать кодовые единицы, представляющие кодовые точки Unicode в кодировке UTF-16. Наиболее часто используемые символы Unicode представляются одной кодовой единицей. Дополнительные символы задаются парами кодовых единиц.

Метод length() возвращает число кодовых единиц для данной строки в кодировке UTF-16. Ниже приведен пример использования данного метода.

```
String greeting = "Hello";
int n = greeting.length(); // Значение n равно 5.
```

Чтобы определить реальную длину, представляющую собой число кодовых точек, надо использовать следующий вызов:

```
int cpCount = greeting.codePointCount(0, greeting.length());
```

Метод s.charAt(n) возвращает кодовую единицу в позиции n, где n находится в интервале от 0 до s.length() - 1. Ниже приведены примеры вызова данного метода.

```
char first = greeting.charAt(0); // Первый символ - 'H'
char last = greeting.charAt(4); // Последний символ - 'o'
```

Для получения i-й кодовой точки надо использовать приведенные ниже выражения.

```
int index = greeting.offsetByCodePoints(0, i);
int cp = greeting.codePointAt(index);
```

Использование кодовых единиц может привести к недоразумениям. Рассмотрим приведенную ниже строку.

```
Z is the set of integers
```

Для представления символа Z используются две кодовые единицы UTF-16. Приведенный ниже вызов метода даст не код пробела, а второй код символа Z. Чтобы избежать возникновения данной проблемы, не следует использовать тип char, так как он представляет символы на слишком низком уровне.

```
char ch = sentence.charAt(i);
```

Если вы хотите просмотреть строку посимвольно, т.е. получить по очереди каждую кодовую точку, вам надо использовать фрагмент кода, подобный приведенному ниже.

```
int cp = sentence.codePointAt(i);
if (Character.isSupplementaryCodePoint(cp)) i += 2;
else i++;
```

Метод codePointAt() определяет, является ли кодовая единица первой или второй частью дополнительного символа, и всегда возвращает корректный результат. Поэтому вы можете организовать просмотр строки и в обратном направлении.

```
i--;
int cp = sentence.codePointAt(i);
if (Character.isSupplementaryCodePoint(cp)) i--;
```

Подстроки

С помощью метода substring() класса String можно выделить подстроку данной строки. Например, в результате выполнения приведенного ниже кода формируется строка "Hel".

```
String greeting = "Hello";
String s = greeting.substring(0, 3);
```

Второй параметр метода `substring()` — это первая кодовая единица, которую *не следует* включать в состав подстроки. В данном примере мы хотим скопировать символы из позиций 0, 1 и 2, поэтому задаем при вызове метода `substring()` значения 0 и 3.

Описанный способ вызова метода `substring()` имеет положительную особенность: подсчет кодовых единиц в подстроке осуществляется исключительно просто. Строка `s.substring(a, b)` всегда содержит `b - a` кодовых единиц. Например, в сформированной выше строке "Hel" содержится `3 - 0 = 3` кодовых единиц.

Изменение строк

В классе `String` отсутствуют методы, которые позволяли бы изменять символы в существующих строках. Если, например, вы хотите изменить строку `greeting` с `"Hello"` на `"Help!"`, то заменить требуемые два символа невозможно. Для разработчиков, использующих язык C, это звучит, по крайней мере, странно. "Как же модифицировать строку?" — спросят они. В Java внести необходимые изменения можно, выполнив конкатенацию подстроки `greeting` и символов `"p!"`

```
greeting = greeting.substring(0, 3) + "p!";
```

В результате текущим значением переменной `greeting` становится строка `"Help!"`.

Поскольку, программируя на языке Java, вы не можете изменять отдельные символы в строке, в документации для описания объектов *String* используется термин *неизменяемые, или немодифицируемые* (immutable). Как число 3 всегда равно 3, так и строка `"Hello"` всегда содержит символы `'H'`, `'e'`, `'l'`, `'l'` и `'o'`. Изменить эти значения невозможно. Однако, как мы только что убедились, можно изменить содержимое строковой *переменной* `greeting` и заставить ее ссылаться на другую строку так же, как числовой переменной, в которой хранится число 3, можно присвоить число 4.

Не снижается ли при этом эффективность? Кажется, было бы намного проще изменять символы, чем создавать новую строку заново. Возможно, это и так. Действительно, неэффективно создавать новую строку посредством конкатенации строк `"Hel"` и `"p!"`. Однако неизменяемые строки имеют одно большое преимущество: компилятор может делать строки *совместно используемыми*.

Чтобы понять этот принцип, представьте, что в совместно используемом пуле находятся разные строки. Строковые переменные указывают объекты в этом пуле. При копировании строки переменной и оригинал, и копия содержат одну и ту же последовательность символов. Логично не прибегать к дублированию строк, а поместить в переменные ссылки одну и ту же область памяти. Одним словом, разработчики языка Java решили, что эффективность совместного использования памяти перевешивает неэффективность редактирования строк путем выделения подстрок и конкатенации.

Посмотрите на свою программу; мы подозреваем, что большую часть времени вы проводите, не изменяя строки, а сравнивая их. Разумеется, бывают случаи, когда непосредственные манипуляции со строками более эффективны. (Одна из таких ситуаций возникает, когда нужно образовать строку из отдельных символов, поступающих из файла или клавиатуры.) Для этих ситуаций в языке Java предусмотрен отдельный класс `StringBuffer`, который будет описан в главе 12. Если же эффективность обработки строк для вас не важна (как это часто случается во многих приложениях на языке Java), вы можете не применять класс `StringBuffer`, а использовать лишь класс `String`.

Когда программисты, работающие на языке C, видят строки в Java-программе, они обычно попадают в тупик, поскольку привыкли рассматривать строки как массивы символов:

```
char greeting[] = "Hello";
```

Это не совсем подходящая аналогия: строка в языке Java больше напоминает указатель *char:

```
char* greeting = "Hello";
```

При замене содержимого `greeting` другой строкой Java-программа выполняет примерно следующее:

```
char* temp = malloc(6);
strncpy(temp, greeting, 4);
strncpy(temp + 4, "!", 2);
greeting = temp;
```

Разумеется, теперь переменная `greeting` указывает на строку `"Help!"`. И даже самые убежденные поклонники языка C должны признать, что синтаксис Java более элегантен, чем последовательность вызовов функции `strncpy()`. А что будет, если мы присвоим строке `greeting` еще одно значение?

```
greeting = "Howdy";
```

Не останется ли занятой ранее выделенная память? К счастью, в языке Java есть механизм автоматической "сборки мусора". Если память больше не нужна, она вскоре будет освобождена.

Если вы программируете на языке C++ и применяете класс `string`, определенный в стандарте ANSI C++, вам будет намного легче работать с объектами `String` в языке Java. Объекты класса `string` в языке C++ также обеспечивают автоматическое выделение и освобождение памяти. Однако строки в языке C++ могут изменяться — отдельные символы в строке можно модифицировать.

Конкатенация

Язык Java, как и большинство языков программирования, дает возможность использовать знак + для объединения (конкатенации) двух строк.

```
String expletive = "Вставка";
String PG13 = "удаленная";
String message = expletive + PG13;
```

Код, приведенный выше, присваивает переменной `message` строку `"Вставкаудаленная"`. (Обратите внимание на отсутствие пробела между словами: знак + объединяет в *точности* те строки, которые были заданы.)

При конкатенации строки со значением, отличным от строкового, это значение преобразовывается в строку. Например, в результате выполнения приведенного ниже кода переменной `rating` присваивается строка `"PG13"`.

```
int age = 13;
String rating = "PG" + age;
```

Это свойство широко используется в операторах вывода; например, приведенное ниже выражение выводит результат выполненных ранее вычислений (обратите внимание на пробел после слова `Ответ`).

```
System.out.println("Ответ " + answer);
```

Проверка совпадения строк

Чтобы проверить, совпадают ли две строки, следует использовать метод `equals()`. Приведенное ниже выражение возвращает значение `true`, если строки `s` и `t` равны между собой, в противном случае возвращается значение `false`.

```
s.equals(t)
```

Заметим, что в качестве `s` и `t` могут быть использованы как переменные, так и константы. Например, следующее выражение вполне допустимо:

```
"Hello!".equals(greeting);
```

Чтобы проверить идентичность строк, игнорируя различие между прописными и строчными буквами, следует использовать метод `equalsIgnoreCase()`.

```
"Hello".equalsIgnoreCase("hello");
```

Для проверки строк на равенство нельзя применять оператор ==! Он лишь определяет, хранятся ли обе строки в одной и той же области памяти. Разумеется, если обе строки хранятся в одном и том же месте, они должны совпадать между собой. Однако вполне возможна ситуация, при которой идентичные строки хранятся в разных местах.

```
String greeting = "Hello"; // Инициализирует переменную
                           // greeting строкой.
if (greeting = "Hello") ...
    // Возможно, это условие истинно.
if (greeting.substring(0, 3) == "Hel") ...
    // Возможно, это условие ложно.
```

Если виртуальная машина всегда обеспечивает совместное использование одинаковых строк, то для проверки их равенства можно применять оператор ==. Однако совместно использовать можно лишь *константы*, а не строки, являющиеся результатами таких операций, как + или `substring`. Следовательно, либо нужно *навсегда* отказаться от проверок строк на равенство с помощью оператора ==, либо вы получите программу, содержащую наихудшую из возможных ошибок, проявляющуюся лишь время от времени, — ошибку, возникновение которой невозможно предсказать.

Если вы привыкли использовать класс `string` в языке C++, будьте особенно внимательны при проверке совпадения строк. В классе `string` оператор == перегружен и позволяет проверять идентичность содержимого строк. Возможно, в языке Java разработчики напрасно отказались от возможности работать со строками как с числовыми значениями, однако это позволило сделать строки похожими на указатели. Разработчики могли переопределить оператор == для строк так, как они это сделали с оператором +. Что ж, каждый язык имеет свои недостатки.

Программисты, создающие продукты на языке C, никогда не используют для проверки строк на равенство оператор ==, вместо этого они вызывают функцию `strcmp()`. Метод `compareTo()` в языке Java представляет собой точный аналог функции `strcmp()`. Можно, конечно, использовать выражения вида

```
if (greeting.compareTo("Help") == 0) ...
```

Однако нам кажется, что применение метода `equals()` делает программу более удобочитаемой.

Класс `String` в языке Java содержит более 50 методов. Многие из них оказались очень полезными и используются довольно часто. Приведенный ниже фрагмент описания API содержит наиболее полезные из них.

Время от времени мы будем включать в текст книги фрагменты описания прикладного программного интерфейса Java (Application Programming Interface — API). Каждый такой фрагмент начинается с имени класса, например `java.lang.String`. В данном случае `java.lang` — это пакет; механизм пакетов будет подробно рассмотрен в главе 4. После имени класса следуют имена конкретных методов и их описание.

Мы также приводим номер версии, в которой был реализован описываемый класс. Если тот или иной метод был добавлен позже, это указывается отдельно.

Обычно в тексте книги не перечисляются все методы отдельного класса; мы ограничиваемся лишь наиболее часто используемыми методами. Полный список методов можно найти в справочной системе.

`java.lang.String` 1.0

- `char charAt(int index)`

  Возвращает символ, расположенный в указанной позиции. Вызывать этот метод следует только в том случае, если вас интересуют низкоуровневые кодовые единицы.

- `int codePointAt(int index)` 5.0

  Возвращает кодовую точку, начало или конец которой находится в указанной позиции.

- `int offsetByCodePoints(int startIndex, int cpCount)` 5.0

  Возвращает индекс кодовой точки, которая определяется `cpCount` относительно `startIndex`.

- `int compareTo(String other)`

  Возвращает отрицательное значение, если данная строка лексикографически предшествует строке `other`, положительное значение — если строка `other` предшествует данной строке, и 0 — если строки идентичны.

- `boolean endsWith(String suffix)`

  Возвращает значение `true`, если строка заканчивается подстрокой `suffix`.

- `boolean equals(Object other)`

  Возвращает значение `true`, если данная строка совпадает со строкой `other`.

- `boolean equalsIgnoreCase(String other)`

  Возвращает значение `true`, если данная строка совпадает со строкой `other` без учета регистра символов.

- `int indexOf(String str)`
- `int indexOf(String str, int fromIndex)`
- `int indexOf(int cp)`
- `int indexOf(int cp, int fromIndex)`

Возвращает индекс начала первой подстроки, совпадающей со строкой `str`, либо индекс указанной кодовой точки. Отсчет начинается с позиции `0` или `formIndex`.
Если указанная подстрока в составе строки отсутствует, возвращается значение, равное –1.

- `int lastIndexOf(String str)`
- `int lastIndexOf(String str, int fromIndex)`
- `int lastindexOf(int cp)`
- `int lastindexOf(int cp, int fromIndex)`

Возвращает начало последней подстроки, равной строке `str`, либо индекс указанной кодовой точки. Отсчет начинается с позиции `0` или `formIndex`. Если указанная подстрока в составе строки отсутствует, возвращается значение, равное –1.

- `int length()`

Возвращает длину строки.

- `int codePointCount(int startIndex, int endIndex)` 5.0

Возвращает число кодовых точек между startIndex и endIndex –1. Половина пары, обозначающей дополнительный индекс, считается как полноправная кодовая точка.

- `String replace(CharSequence oldString, CharSequence newString)`

Возвращает новую строку, которая получается путем замены всех подстрок, соответствующих `oldString`, на строку `newString`. В качестве параметров `CharSequence` могут выступать объекты `String` или `StringBuilder`.

- `boolean startWith(String prefix)`

Возвращает значение `true`, если строка начинается подстрокой `prefix`.

- `String substring(int beginIndex)`
- `String substring(int beginIndex, int endIndex)`

Возвращает новую строку, состоящую из всех кодовых единиц, начиная с позиции `BeginIndex` и заканчивая концом строки или позицией `endIndex -1`.

- `String toLowerCase()`

Возвращает новую строку, состоящую из всех символов исходной строки. Отличие между исходной и результирующей строкой состоит в том, что все буквы преобразуются в нижний регистр.

- `String toUpperCase()`

Возвращает новую строку, состоящую из всех символов исходной строки. Отличие между исходной и результирующей строкой состоит в том, что все буквы преобразуются в верхний регистр.

- `String trim()`

Возвращает новую строку, из которой исключены все предшествующие и завершающие *пробелы*.

Интерактивная документация по API

Как вы уже знаете, класс `String` имеет много методов. Более того, в стандартной библиотеке существует несколько сотен классов, содержащих огромное количество методов. Совершенно невозможно запомнить всю информацию, которая может понадобиться при написании программ. Следовательно, очень важно уметь пользоваться интерактивной справочной системой, содержащей документацию об API, и находить нужные классы и методы. Документация является составной частью набора инструментальных средств JDK. Она представлена в формате HTML. Обратитесь с помощью вашего браузера к документу `docs/api/index.html` в каталоге, в котором установлен JDK. Вы увидите информацию, показанную на рис. 3.2.

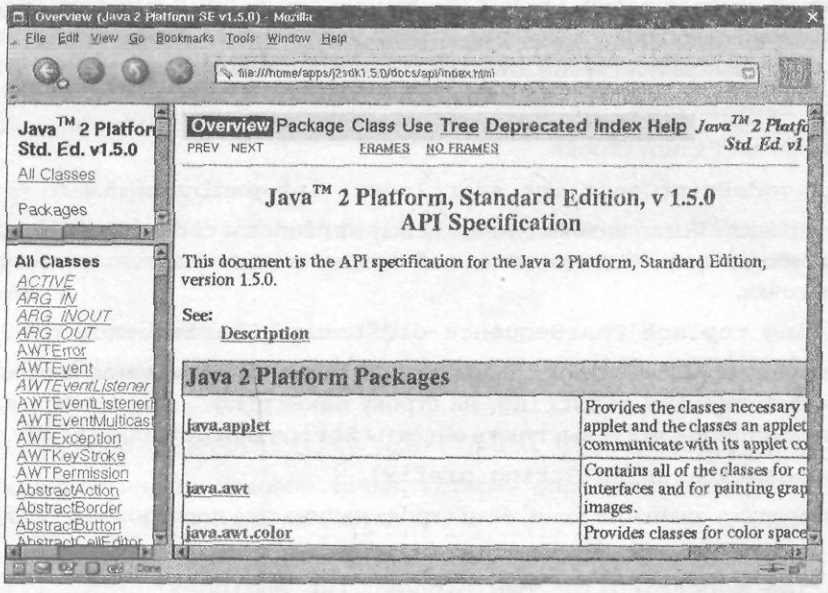


Рис. 3.2. Окно с документацией по API

Окно браузера разделено на три фрейма. В небольшом фрейме в верхнем левом углу отображаются имена всех доступных пакетов. Под ним во фрейме побольше перечислены все классы. Щелкните мышью на любом из имен классов, и информация об этом классе будет показана в большом фрейме, расположенном справа (рис. 3.3). Например, чтобы получить дополнительные сведения о методах класса `String`, прокрутите содержимое левого нижнего фрейма, пока не увидите ссылку `String`, и щелкните на ней.

Затем прокрутите содержимое правого фрейма до тех пор, пока не увидите краткое описание всех методов. Методы расположены в алфавитном порядке (рис. 3.4). Щелкните на имени интересующего вас метода, чтобы получить его детальное описание (рис. 3.5). Например, если вы щелкнете на ссылке `compareToIgnoreCase`, то получите описание метода с этим именем.

Сразу сделайте закладку в браузере, указывающую на документ `docs/api/index.html`.

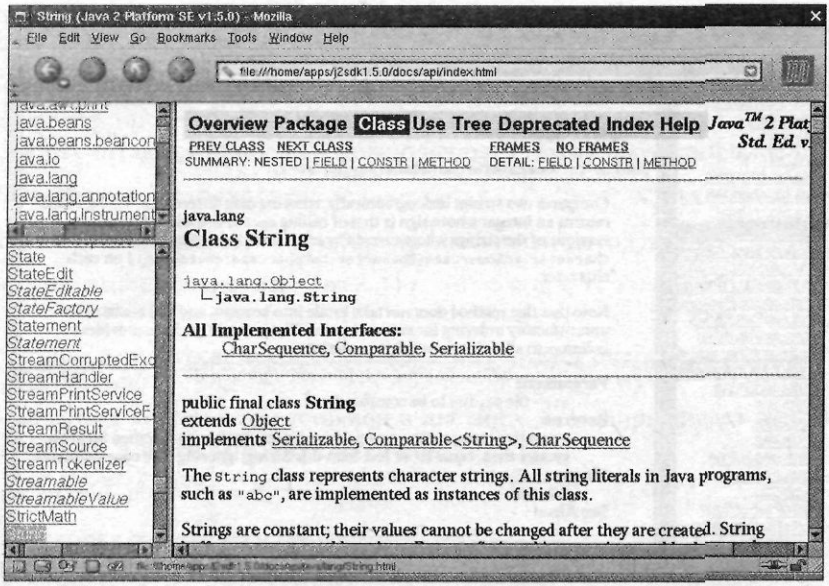


Рис. 3.3. Описание класса String

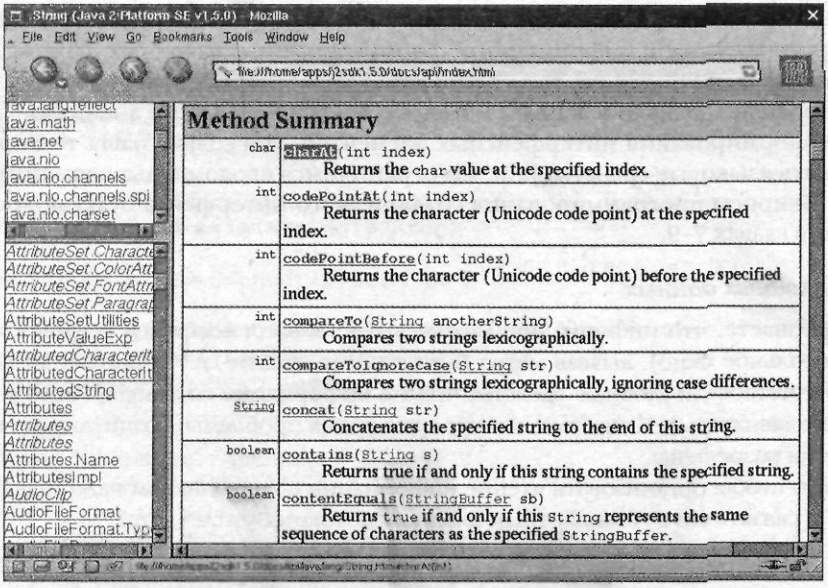


Рис. 3.4. Перечень методов класса `String`

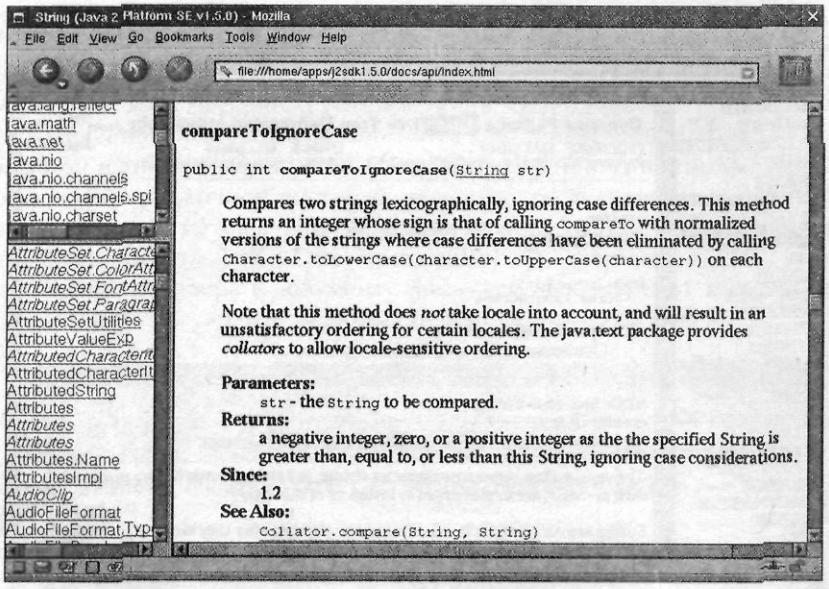


Рис. 3.5. Подробное описание метода `compareToIgnoreCase()` класса `String`

Ввод и вывод

Для того чтобы немного "оживить" программы, рассматриваемые в качестве примеров, организуем ввод информации и форматирование выходных данных. В современных приложениях для ввода используются средства графического пользовательского интерфейса, однако в данный момент вы еще не обладаете знаниями, достаточными для формирования интерфейсных элементов. Поскольку наша текущая цель – лучше узнать языковые средства Java, мы ограничимся вводом и выводом посредством консоли. Вопросы программирования графического интерфейса будут подробно рассмотрены в главах 7–9.

Чтение входных данных

Вы уже знаете, что информацию можно вывести на стандартное устройство вывода (т.е. в консольное окно), вызвав метод `System.out.println()`. Однако до появления JDK 5.0 отсутствовали удобные средства чтения информации со стандартного устройства ввода (т.е. клавиатуры). К счастью, в настоящее время проблемы, возникающие при этом, практически разрешены.

Для того чтобы организовать чтение информации с консоли, вам надо создать объект `Scanner` и связать его со стандартным входным потоком `System.in`.

```
Scanner in = new Scanner(System.in);
```

Сделав это, вы получите в свое распоряжение многочисленные методы класса `Scanner`, предназначенные для чтения входных данных. Например, метод `nextLine()` обеспечивает прием строки текста.

```
System.out.print("Как вас зовут? ");
String name = in.nextLine();
```

В данном случае мы использовали метод `nextLine()`, потому что входная строка может содержать пробелы. Для того чтобы прочитать одно слово (разделителями между словами считаются пробелы), можно использовать следующий вызов:

```
String firstName = in.next();
```

Для чтения целочисленного значения предназначен метод `nextInt()`.

```
System.out.print("Сколько вам лет? ");
int age = in.nextInt();
```

Как нетрудно догадаться, метод `nextDouble()` читает очередное число в формате с плавающей точкой.

Программа, код которой представлен в листинге 3.2, запрашивает имя пользователя и его возраст, а затем выводит сообщение типа

```
Cay, в следующем году вам будет 46
```

В первой строке содержится выражение

```
import java.util.*;
```

Класс `Scanner` принадлежит пакету `java.util` package. Если вы собираетесь использовать в программе класс, не содержащийся в базовом пакете `java.lang`, вам надо включить в состав кода директиву `import`. Подробно пакеты и директива `import` будут рассмотрены в главе 4.

Листинг 3.2. Содержимое файла `InputTest.java`

```
import java.util.*;

public class InputTest
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      // Получение первой строки входных данных
      System.out.print("Как вас зовут? ");
      String name = in.nextLine();

      // Получение целочисленного значения
      System.out.print("Сколько вам лет? ");
      int age = in.nextInt();

      // Отображение информации в консольном окне
      System.out.println("name +
          ", в следующем году вам будет " + (age + 1));
   }
}
```

Если вы не имеете возможности работать с JDK 5.0 или более поздними версиями, то организовать ввод данных будет несколько труднее. Вам придется использовать диалоговое окно, показанное на рис. 3.6.

```
String inut = JOptionPane.showInputDialog(promptString);
```

Возвращаемое значение представляет собой строку, введенную пользователем.

Например, следующая строка кода позволяет запросить имя пользователя вашей программы:

```
String name = JOptionPane.showInputDialog("Как вас зовут?");
```

Чтобы считать число, нужно выполнить более сложную работу. Метод `JOptionPane.showInputDialog()` возвращает строку, а не число. Для преобразования этой строки в число нужно использовать метод `Integer.parseInt()` или `Double.parseDouble()`. Пример такого преобразования приведен ниже.

```
String input = JOptionPane.showInputDialog("Сколько вам лет?");
int age = Integer.parseInt(input);
```

Если пользователь введет число `45`, то строковой переменной `input` будет присвоена строка `"45"`. Метод `Integer.parseInt()` преобразовывает строку в соответствующее число, т.е. 45.

Класс `JOptionPane` принадлежит пакету `javax.swing`, поэтому вам надо включить в состав исходного файла следующее выражение:

```
import javax.swing.*;
```

И наконец, если ваша программа вызывает метод `JOptionPane.showInputDialog()`, вам надо завершить ее вызовом `System.exit(0)`. Дело в том, что при создании диалогового окна формируется новый управляющий поток. Когда выполнение метода `main()` оканчивается, автоматического завершения потока не происходит. Для того чтобы прекратить работу всех потоков, надо вызвать метод `System.exit()`. Дополнительную информацию о потоках можно найти в главе 1 второго тома. Следующая программа обеспечивает те же возможности, что и программа, представленная в листинге 3.2, но не использует средства JDK 5.0.

```
import javax.swing.*;
public class InputTest
{
   public static void main(String[] args)
   {
      String name = JOptionPane.showInputDialog(

         "Как вас зовут?");
      String input = JOptionPane.showInputDialog(

         "Сколько вам лет?");
      int age = Integer.parseInt(input);
      System.out.println(name +

         ", в следующем году вам будет " + (age + 1));
      System.exit(0);
   }
}
```

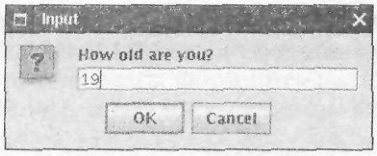


Рис. 3.6. Диалоговое окно для ввода информации

java.util.Scanner 5.0

- `Scanner(InputStream in)`

  Создает объект `Scanner` на базе заданного входного потока.

- `String nextLine()`

  Читает очередную строку.

- `String next()`

  Читает очередное слово (разделителями между словами считаются пробелы).

- `int nextInt()`
- `double nextDouble()`

  Читает очередную последовательность символов, представляющую целое число или число с плавающей точкой, и выполняет соответствующее преобразование.

- `boolean hasNext()`

  Проверяет, существует ли во входном потоке еще одно слово.

- `boolean hasNextInt()`
- `boolean hasNextDouble()`

  Проверяет, существует ли во входном потоке последовательность символов, целое число или число с плавающей точкой.

javax.swing.JOptionPane 1.2

- `static String showInputDialog(Object message)`

  Выводит на экран диалоговое окно с сообщением, полем ввода и кнопками OK и Cancel. Метод возвращает строку, введенную пользователем.

java.lang.System 1.0

- `static void exit(int status)`

  Прекращает работу виртуальной машины и передает операционной системе код завершения. По умолчанию ненулевой код завершения соответствует возникновению ошибки.

Форматирование выходных данных

Число x можно вывести на консоль с помощью выражения `System.out.println(x)`. В результате на экране отобразится число с максимальным количеством значащих цифр,

допустимых для данного типа. Например, в результате выполнения приведенного ниже фрагмента кода на экран будет выведено число 3333.3333333333335.

```
double x = 10000.0 / 3.0;
System.out.print(x);
```

В ряде случаев это создает проблемы. Так, например, если вы хотите вывести на экран сумму в долларах и центах, большое количество цифр затруднит восприятие.

До появления JDK 5.0 процесс форматирования чисел был сопряжен с определенными трудностями. В JDK 5.0 был реализован метод `printf()`, привычный всем программистам, использующим язык C. Например, с помощью приведенного ниже выражения мы можем вывести значение x в виде числа, размер поля которого составляет 8 цифр, а дробная часть равна двум цифрам. (Число цифр дробной части называют также точностью.)

```
System.out.printf("%8.2f", x);
```

В результате на экран будет выведено, не считая ведущих пробелов, семь символов:

```
3333.33
```

Метод `printf()` позволяет задавать произвольное число параметров. Пример вызова с несколькими параметрами приведен ниже.

```
System.out.printf("%s, в следующем году вам будет %d", name, age);
```

Каждый *спецификатор формата*, начинающийся с символа `%`, заменяется соответствующим параметром. *Символ преобразования*, которым завершается спецификатор формата, задает тип форматируемого значения: `f` — число с плавающей точкой; `s` — строка; `d` — десятичное число. Символы преобразования описаны в табл. 3.5.

Таблица 3.5. Символы преобразования для метода `printf()`

Символ преобразования	Тип	Пример
`d`	Десятичное целое	`159`
`x`	Шестнадцатеричное целое	`9f`
`o`	Восьмеричное целое	`237`
`f`	Число с фиксированной или плавающей точкой	`15.9`
`e`	Число с плавающей точкой в экспоненциальной форме	`1.59e+01`
`g`	Число с плавающей точкой в общем формате (чаще всего используется формат `e` или `f`, в зависимости от того, какой из них дает более короткую запись)	
`a`	Шестнадцатеричное представление числа с плавающей точкой	`0x1.fccdp3`
`s`	Строка	`Hello`
`c`	Символ	`H`
`b`	Логическое значение	`true`
`h`	Хэш-код	`42628b2`
`tx`	Дата и время	См. табл. 3.7
`%`	Символ процента	`%`
`n`	Разделитель строк, зависящий от платформы	

В составе спецификатора формата могут присутствовать флаги, управляющие форматом выходных данных. Назначение всех флагов описано в табл. 3.6. Например, запятая, используемая в качестве флага, формирует разделитель групп. Так, в результате выполнения приведенного ниже выражения на экран будет выведена строка 3,333.33.

```
System.out.printf("%,.2f", 10000.0 / 3.0);
```

В одном спецификаторе формата можно использовать несколько флагов, например последовательность символов "%, (.2f" указывает на то, что при выводе будут использованы разделители групп, а отрицательные числа будут помещены в скобки).

Преобразование s можно использовать для форматирования любого объекта. Если этот объект реализует интерфейс `Formattable`, вызывается метод `formatTo()`. В противном случае для преобразования объекта в строку применяется метод `toString()`. Метод `toString()` будет обсуждаться в главе 5, а интерфейсы — в главе 6.

Для создания форматированной строки без вывода ее можно использовать статический метод `String.format()`.

```
String message = String.format("%s, в следующем году вам будет %d",
   name, age);
```

Таблица 3.6. Флаги для метода `printf()`

Флаг	Назначение	Пример
+	Выводит знак не только для отрицательных, но и для положительных чисел	+3333.33
Пробел	Добавляет пробел перед положительными числами	| 3333.33|
0	Выводит ведущие нули	003333.33
-	Выравнивает поле по левому краю	|3333.33 |
(	Помещает отрицательные числа в скобки	(3333.33)
,	Задает использование разделителя групп	3,333.33
# (для формата f)	Всегда отображает десятичную точку	3,333.
# (для формата x или o)	Добавляет префикс 0x или 0	0xcafe
^	Задает преобразование в верхний регистр	0XCAFE
$	Определяет индекс параметра, предназначенного для форматирования. Например, выражение %1$d %1$x указывает на то, что первый параметр должен быть выведен в десятичном формате, а затем он же представлен в шестнадцатеричном виде	159 9F
<	Задает форматирование того же значения, к которому относился предыдущий спецификатор. Например, выражение %d %<x указывает на то, что одно и то же значение должно быть представлено как в десятичном, так и в шестнадцатеричном виде	159 9F

Несмотря на то, что тип Date будет подробно рассмотрен лишь в главе 4, нам, чтобы закончить разговор о методе printf(), желательно подробно рассмотреть средства форматирования даты и времени. Для этой цели используется последовательность из двух символов, начинающаяся с буквы t, за которой следует одна из букв, приведенных в табл. 3.7. Пример такого форматирования приведен ниже.

```
System.out.printf("%tc", new Date());
```

В результате выполнения данного выражения выводится текущая дата и время.

```
Mon Feb 09 18:05:19 PST 2004
```

Как видно в табл. 3.7, некоторые форматы предполагают отображение лишь отдельных компонентов даты — дня или месяца. Было бы неразумно многократно задавать дату лишь для того, чтобы отформатировать различные ее элементы. По этой причине в строке, определяющей формат, может задаваться индекс форматируемого параметра. Индекс должен следовать непосредственно за символом % и завершаться знаком $. Пример использования индекса приведен ниже.

```
System.out.printf("%1$s %2$tB %2$te, %2$tY", "Дата:", new Date());
```

Таблица 3.7. Символы для форматирования даты и времени

Символ преобразования	Выходная информация	Пример
c	Полные сведения о дате и времени	Mon Feb 09 18:05:19 PST 2004
F	Дата в формате ISO 8601	2004-02-09
D	Дата в формате, принятом в США (месяц/день/год)	02/09/2004
T	Время при 24-часовом цикле	18:05:19
r	Время при 12-часовом цикле	06:05:19 pm
R	Время при 24-часовом цикле (секунды не выводятся)	18:05
Y	Год, представленный четырьмя цифрами (при необходимости отображаются ведущие нули)	2004
y	Последние две цифры года (при необходимости отображаются ведущие нули)	04
C	Первые две цифры года (при необходимости отображаются ведущие нули)	20
B	Полное название месяца	February
b или h	Сокращенное название месяца	Feb
m	Месяц, представленный двумя цифрами (при необходимости отображаются ведущие нули)	02
d	День, представленный двумя цифрами (при необходимости отображаются ведущие нули)	09
e	День, представленный одной или двумя цифрами (ведущие нули не отображаются)	9
A	Полное название дня недели	Monday
a	Сокращенное название дня недели	Mon

Окончание табл. 3.7

Символ преобразования	Выходная информация	Пример
j	День в году, представленный тремя цифрами (при необходимости отображаются ведущие нули); номер дня может лежать в пределах от 001 до 366	069
H	Номер часа, представленный двумя цифрами (при необходимости отображаются ведущие нули); номер часа может лежать в пределах от 00 до 23	18
k	Номер часа, представленный одной или двумя цифрами (ведущие нули не отображаются); номер часа может лежать в пределах от 0 до 23	18
I	Номер часа, представленный двумя цифрами (при необходимости отображаются ведущие нули); номер часа может лежать в пределах от 01 до 12	06
l	Номер часа, представленный одной или двумя цифрами (ведущие нули не отображаются); номер часа может лежать в пределах от 1 до 12	6
M	Число минут, представленное двумя цифрами (при необходимости отображаются ведущие нули)	05
S	Число секунд, представленное двумя цифрами (при необходимости отображаются ведущие нули)	19
L	Число миллисекунд, представленное двумя цифрами (при необходимости отображаются ведущие нули)	047
N	Число наносекунд, представленное девятью цифрами (при необходимости отображаются ведущие нули)	047000000
P	AM и PM отображаются символами верхнего регистра	PM
p	AM и PM отображаются символами нижнего регистра	pm
z	Смещение относительно гринвичского времени (GMT) согласно RFC 822	-0800
Z	Временной пояс	PST
s	Число секунд с 1970-01-01 00:00:00 GMT	1078884319
E	Число миллисекунд с 1970-01-01 00:00:00 GMT	1078884319047

В результате обработки данного выражения будет выведена следующая строка:

```
Дата: February 9, 2004
```

Вы также можете использовать флаг <, который означает, что форматированию подлежит тот же параметр, который был сформатирован последним. Так, приведенное ниже выражение дает тот же результат, что и рассмотренное ранее.

```
System.out.printf("%s %tB %<te, %<tY", "Дата:", new Date());
```

Индексы начинаются с единицы. Так, выражение `%1$` задает форматирование первого параметра. Это сделано для того, чтобы избежать конфликтов с флагом `0`.

В данном разделе были рассмотрены не все особенности метода `printf()`. На рис. 3.7 показана диаграмма, которая строго задает синтаксис спецификатора формата.

Ряд правил форматирования зависит от специфических условий конкретной страны или региона. Так, например, в Германии разделителем групп является не запятая, а точка, а вместо `Monday` выводится имя `Montag`. Вопросы интернационализации приложений будут рассмотрены во втором томе.

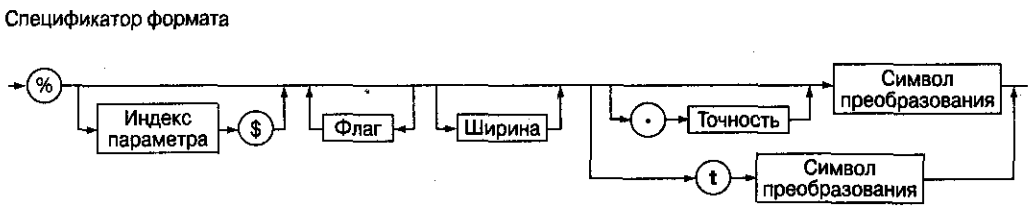


Рис. 3.7. Правила записи спецификатора формата

Если вы работаете с одной из версий, предшествующих JDK 5.0, вместо метода `printf()` вам придется использовать классы `NumberFormat` и `DateFormat`.

Поток управления

В языке Java, как и в любом другом языке программирования, есть условные операторы и циклы. Начнем с условных операторов, а затем перейдем к циклам. Рассмотрение потока управления мы завершим довольно неуклюжим оператором `switch`, который можно применять при проверке большого количества значений одного выражения.

Поток управления в языке Java построен точно так же, как и в языках C и C++, за исключением двух особенностей. В нем нет оператора `goto`, однако существует версия оператора `break` с метками, который можно использовать для выхода из вложенного цикла (в языке C для этого пришлось бы применять оператор `goto`). Кроме того, в JDK 5.0 реализован вариант оператора for, который не имеет аналогов в C или C++. Его можно сравнить с оператором `foreach` в C#.

Блоки

Перед изучением управляющих структур нам необходимо рассмотреть *блоки*.

Блок, или составной оператор, — это произвольное количество простых операторов языка Java, заключенных в фигурные скобки. Блоки определяют область видимости переменных. Блоки могут быть *вложенными* один в другой. Ниже приведен пример блока, вложенного в другой блок в методе `main()`.

```
public static void main(String[] args)
{
   int n;
   ...
   {
      int k;
      ...
```

```
   } // Переменная k определена только в этом блоке.
}
```

В языке Java невозможно объявить переменные с одним именем в двух вложенных блоках. Например, приведенный ниже фрагмент кода содержит ошибку и не будет скомпилирован.

```
public static void main(String[] args)
{
   int n;
   ...
   {
      int k;
      int n; // Ошибка -- невозможно переопределить переменную
             // n во внутреннем блоке.
      ...
   }
}
```

В языке C++ переменные во вложенных блоках можно переопределять. Внутреннее определение маскирует внешнее. Это может привести к ошибкам, поэтому в языке Java подобный подход не реализован.

Условные выражения

Условный оператор в языке Java имеет следующий вид:

```
if (условие) оператор
```

Условие должно указываться в скобках.

В языке Java, как и в большинстве других языков программирования, часто необходимо выполнять много операторов в зависимости от выполнения одного условия. В этом случае используется *блок*, формируемый следующим образом:

```
{
   Выражение_1;
   Выражение_2;

   ...
}
```

Например,

```
if (yourSales >= target)
{
   performance = "Удовлетворительно";
   bonus = 100;
}
```

В этом фрагменте все операторы, помещенные в фигурные скобки, будут выполнены, если значение переменной `yourSales` больше значения переменной `target` или равно ему (рис. 3.8).

Блок (иногда называемый *составным оператором*) позволяет включать несколько (простых) операторов в любую структуру языка Java, которая в ином случае состояла бы лишь из одного (простого) оператора.

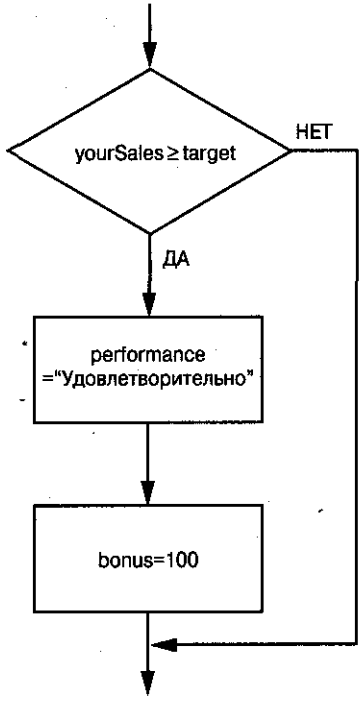


Рис. 3.8. Блок-схема, иллюстрирующая действие оператора `if`

Условное выражение в языке Java может также иметь приведенную ниже форму. Его действие показано на рис. 3.9.

```
if (условие) оператор_1 else оператор_2
```

Например:

```
if (yourSales >= target)
{
   performance = "Удовлетворительно");
   bonus = 100 + 0.01 * (yourSales - target);
}
else
{
   performance = "Неудовлетворительно";
   bonus = 0;
}
```

Часть `else` не является обязательной. Она объединяется с ближайшим оператором `if`. Таким образом, в следующем фрагменте кода оператор `else` соответствует второму оператору `if`:

```
if (x <= 0) if (x == 0) sign = 0; else sign = -1;
```

В программах часто встречаются также операторы типа `if ... else if ...` (рис. 3.10). Пример такой языковой конструкции приведен ниже.

```
if (yourSales >= 2 * target)
{
   performance = "Великолепно";
   bonus = 1000;
}
else if (yourSales >= 1.5 * target)
{
   performance = "Хорошо";
   bonus = 500;
}
else if (yourSales >= target)
{
   performance = "Удовлетворительно";
   bonus = 100;
}
else
{
   System.out.println("Вы уволены");
}
```

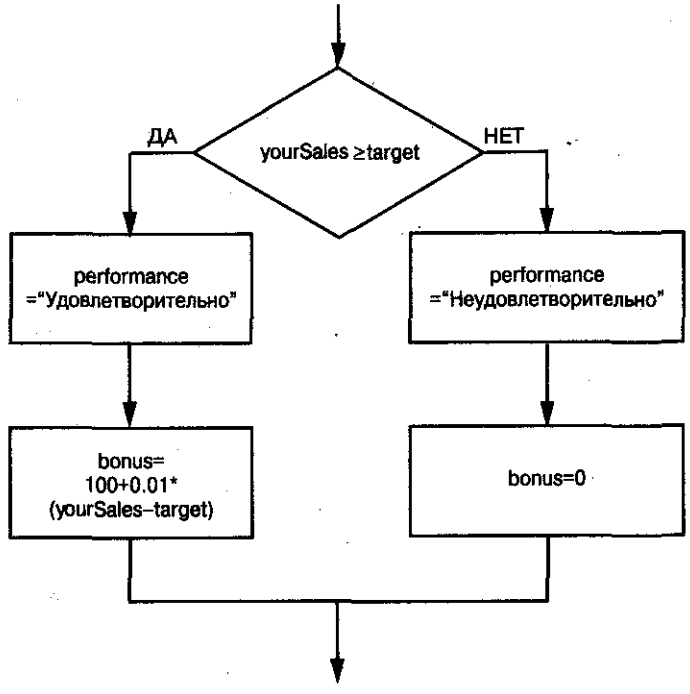


Рис. 3.9. Блок-схема, демонстрирующая действие оператора `if/else`

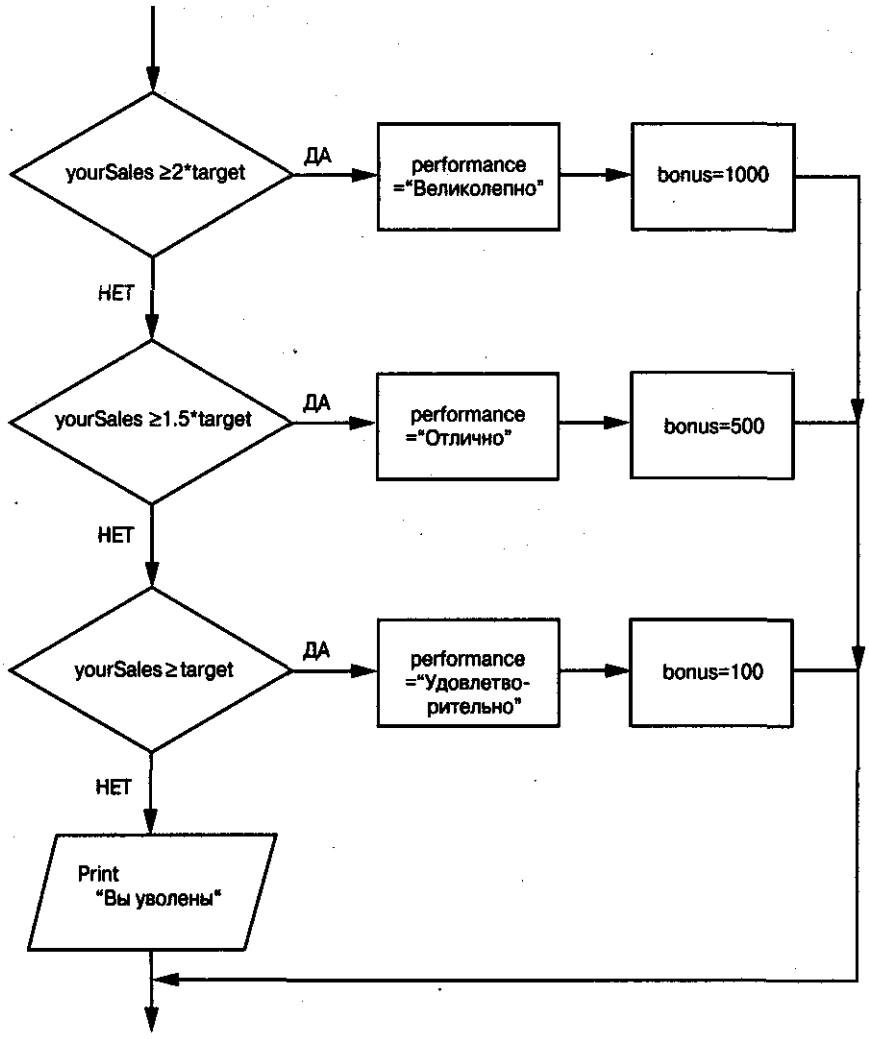


Рис. 3.10. Блок-схема, демонстрирующая действие оператора `if/else if`

Неопределенные циклы

Цикл `while` обеспечивает выполнение выражения (или группы операторов, составляющих блок) до тех пор, пока условие равно `true`. Данный цикл записывается в следующем виде:

```
while (условие) оператор
```

Тело цикла `while` не будет выполнено ни разу, если его условие изначально равно `false` (рис. 3.11).

В листинге 3.3 показана программа, подсчитывающая, сколько лет надо вносить деньги на счет, чтобы накопить заданную сумму. Считается, что каждый год вносится одна и та же сумма и процентная ставка не изменяется.

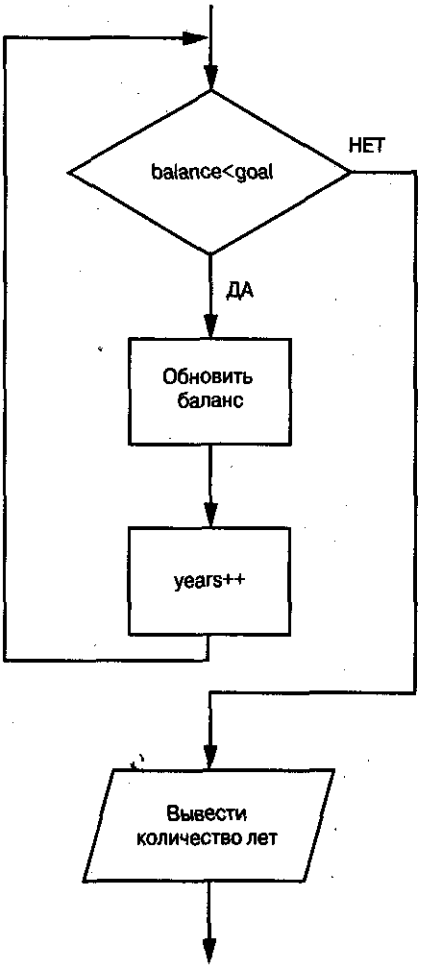


Рис. 3.11. Блок-схема оператора while

В этом примере мы увеличиваем счетчик и обновляем в теле цикла сумму, накопленную к данному моменту, пока общий итог не превысит заданную величину.

```
while (balance < goal)
{
   balance += payment;
   double interest = balance * interestRate / 100;
   balance += interest;
   years++;
}
System.out.println(years + " years.");
```

(Не полагайтесь на эту программу при планировании своей пенсии. Мы отбросили такие детали, как инфляция и ожидаемая продолжительность вашей жизни.)

Условие цикла while проверяется в самом начале. Следовательно, возможна ситуация, при которой код, содержащийся в блоке, не будет выполнен никогда. Если вы хотите, чтобы блок выполнялся хотя бы один раз, проверку условия нужно перенести в конец. Это можно сделать с помощью цикла do/while, который записывается следующим образом:

```
do оператор while (условие);
```

Условие проверяется лишь после выполнения тела цикла. Затем тело цикла повторяется, вновь проверяет условие и т.д. Например, код в листинге 3.4 вычисляет новый баланс вашего счета, а затем спрашивает вас, не собираетесь ли вы на заслуженный отдых.

```
do
{
   balance += payment;
   double interest = balance * interestRate / 100;
   balance += interest;
   year++;
   // Вывести текущий баланс
   ...
   // Спросить, не собирается ли пользователь на пенсию,

   // и получить ответ.
   ...
}
while (input.equals("N"));
```

Если пользователь отвечает N", цикл повторяется (рис. 3.12). Эта программа является хорошим примером применения циклов, которые нужно выполнить хотя бы один раз.

Листинг 3.3. Содержимое файла Retirement.java

```
import java.util.*;

public class Retirement
{
    public static void main(String[] args)
    {
       // Чтение входных данных
       Scanner in = new Scanner(System.in);

       System.out.print(
          "Какая сумма вам необходима?");
       double goal = in.nextDouble();

       System.out.print(
           "Сколько вы можете вносить в год? ");
       double payment = in.nextDouble();

       System.out.print("Процентная ставка: ");
       double interestRate = in.nextDouble();

       double balance = 0;
       int years = 0;
```

```
      // Баланс обновляется до тех пор, пока не будет
      // достигнута требуемая сумма.
      while (balance < goal)
      {
         // Добавление ежегодного взноса и процентов
         balance += payment;
         double interest = balance * interestRate / 100;
         balance += interest;
         years++;
      }

      System.out.println(
         "Вы можете уйти на отдых через " + years + " лет.");
   }
}
```

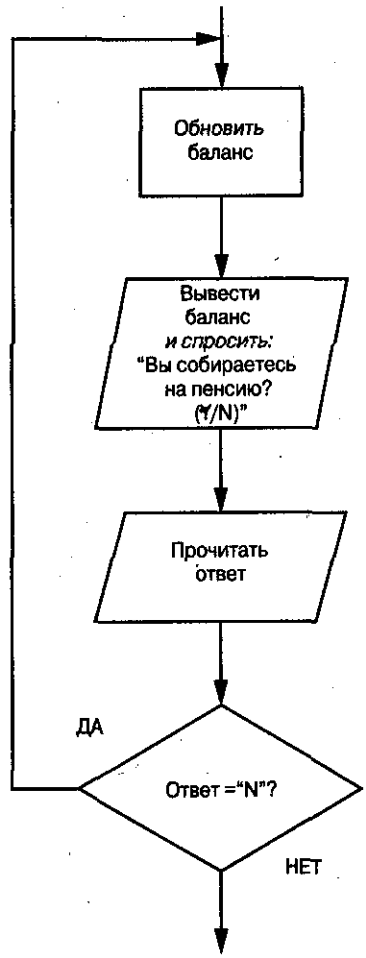


Рис. 3.12. Блок-схема, поясняющая работу оператора do/while

Листинг 3.4. Содержимое файла `Retirement2.java`

```
import java.util.*;

public class Retirement2
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      System.out.print(
         "Сколько вы можете вносить в год? ");
      double payment = in.nextDouble();

      System.out.print("Процентная ставка: ");
      double interestRate = in.nextDouble();

      double balance = 0;
      int year = 0;

      String input;

      // Обновлять баланс, пока пользователь вводит N
      do
      {
         // Добавление ежегодного взноса и процентов
         balance += payment;
         double interest = balance * interestRate / 100;
         balance += interest;

         year++;

         // Вывод текущего баланса
         System.out.printf(
            "Через %d, у вас будет %,.2f%n",
            year, balance);

         // Запрос, готов ли пользователь прекратить работу
         System.out.print("Уходите на отдых? (Y/N) ");
         input = in.next();
      }
      while (input.equals("N"));
   }
}
```

Определенные циклы

Цикл `for` — очень распространенная языковая конструкция. В ней число повторений контролируется переменной, выполняющей роль счетчика и обновляемой на каждой итерации. Приведенный ниже цикл выводит на экран числа от 1 до 10. Его выполнение показано на рис. 3.13.

```
for (int i = 1; i <= 10; i++)
   System.out.println(i);
```

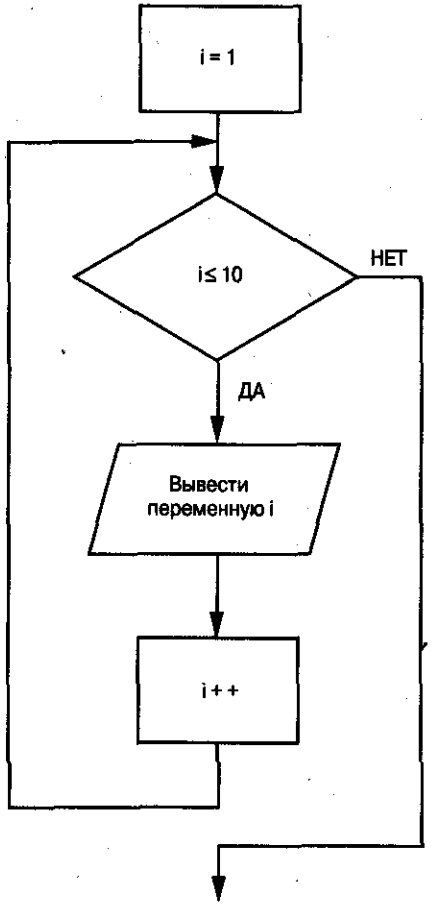


Рис. 3.13. Блок-схема, поясняющая выполнение оператора `for`

Первый элемент оператора `for` обычно выполняет инициализацию счетчика, второй формулирует условие выполнения тела цикла, а третий определяет способ обновления счетчика.

Хотя в языке Java, как и в языке C++, элементами оператора цикла `for` могут быть практически любые операторы, существуют неписанные правила, согласно которым все три элемента оператора `for` должны только инициализировать, проверять и обновлять один и тот же счетчик. Если не придерживаться этих правил, полученный код станет совершенно непригоден для чтения.

Данные правила не слишком сковывают инициативу программиста. Даже придерживаясь их, можно сделать очень многое, например реализовать цикл с убывающим счетчиком.

```
for (int i = 10; i > 0; —i)
   System.out.prinln("Обратный отсчет ..." + i);
System.out.println("Старт!);
```

Будьте осторожны, проверяя в цикле равенство двух чисел с плавающей точкой. Представленный ниже цикл может никогда не завершиться.

```
for (double x = 0; x != 10; x += 0.1) ...
```

Из-за ошибок округления окончательный результат никогда не будет достигнут. Например, в приведенном выше цикле переменная x изменит свое значение с 9.999999999999998 на 10.099999999999998. Так произойдет потому, что для числа 0.1 не существует точного двоичного представления.

При объявлении переменной в первой части оператора `for` *ее область видимости* простирается до конца тела цикла.

```
for (int i = 1; i <= 10; i++)
{
   ...
}
// Здесь переменная i больше не определена.
```

Если переменная определена в операторе `for`, ее нельзя использовать вне этого цикла. Следовательно, если вы хотите использовать конечное значение счетчика вне цикла `for`, эту переменную нужно объявлять до начала цикла!

```
int i;
for (i = 1; i <= 10; i++)
{
   ...
} // Здесь переменная i по-прежнему доступна.
```

С другой стороны, можно объявить переменные, имеющие одинаковое имя в разных циклах `for`.

```
for (int i = 1; i <= 10; i++)
{
   ...
}
   ...
for (int i = 11; i <= 20; i++) // Переопределение переменной i.
{
   ...
}
```

Действия, выполняемые посредством цикла `for`, можно реализовать с помощью цикла `while`. Приведенные ниже два фрагмента кода эквивалентны.

```
for (int i = 10; i > 0; i--)
   System.out.println("Обратный отсчет . . . " + i);
```

и

```
int i = 10;
while (i > 0)
{
   System.out.println("Обратный отсчет . . . " + i);
   i--;
}
```

В листинге 3.5 показан типичный пример использования цикла for.

Эта программа вычисляет вероятность выигрыша в лотерее. Например, если нужно угадать 6 номеров из 50, количество возможных вариантов равно (50×49×48×47×46×45)/(1×2×3×4×5×6), так что шанс выиграть равен 1 из 15890700. Удачи!

В общем случае, если нужно угадать k номеров из n, количество возможных вариантов равно следующему выражению:

$$(n \times (n-1) \times (n-2) \times \ldots \times (n-k+1)) / (1 \times 2 \times 3 \times \ldots \times k)$$

Оно вычисляется с помощью следующего цикла for:

```
int lotteryOdds = 1;
for (int i = 1; i <= k; i++)
   lotteryOdds = lotteryOdds * (n - i + 1) / i;
```

В конце данной главы будет рассмотрен "обобщенный цикл for", называемый также циклом "for each". Эта языковая конструкция была введена в JDK 5.0.

Листинг 3.5. Содержимое файла LotteryOdds.java

```
import java.util.*;

public class LotteryOdds
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      System.out.print(
         "Сколько номеров? ");
      int k = in.nextInt();

      System.out.print(
          "Наибольший номер? ");
      int n = in.nextInt();

       /*
        Вычисление выражения
        n * (n - 1) * (n - 2) * . . . * (n - k + 1)
        ------------------------------------------
        1 * 2 * 3 * . . . * k
      */

      int lotteryOdds = 1;
      for (int i = 1; i <= k; i++)
         lotteryOdds = lotteryOdds * (n - i + 1) / i;

      System.out.println(
        "Ваши шансы 1 из " + lotteryOdds +
        ". Успехов!");
   }
}
```

Многовариантное ветвление — оператор switch

Конструкция `if/else` может оказаться неудобной, если вам необходимо реализовать выбор из многих вариантов. В языке Java есть оператор `switch`, эквивалентный одноименному оператору, используемому в языках C и C++.

Например, программируя выбор из четырех альтернативных вариантов (рис. 3.14), можно использовать следующий код:

```
Scanner in = new Scanner(System.in);
System.out.print("Select an option (1, 2, 3, 4) ");
int choice = in.nextInt();
switch (choice)
{
   case 1:
      ...
      break;
   case 2:
      ...
      break;
   case 3:
      ...
      break;
   case 4:
      ...
      break;
   default:
      // Неверный выбор.
      ...
      break;
}
```

Выполнение начинается с метки `case`, соответствующей значению переменной `choice`, и продолжается до очередного оператора `break` или конца оператора `switch`. Если ни одна метка не совпадает со значением переменной, выполняется раздел `default` (если он предусмотрен).

Заметим, что метка `case` должна быть целочисленной. Нельзя проверять строки. Например, в следующем фрагменте кода сделана ошибка:

```
String input = ...;
switch (input) // ОШИБКА
{
   case "A":   // ОШИБКА
      ...
      break;
   ...
}
```

Если вы забудете добавить в конце раздела `case` оператор `break`, возможно последовательное выполнение нескольких разделов `case`. Совершенно очевидно, что эта ситуация чревата ошибками, поэтому мы никогда не используем оператор `switch` в своих программах.

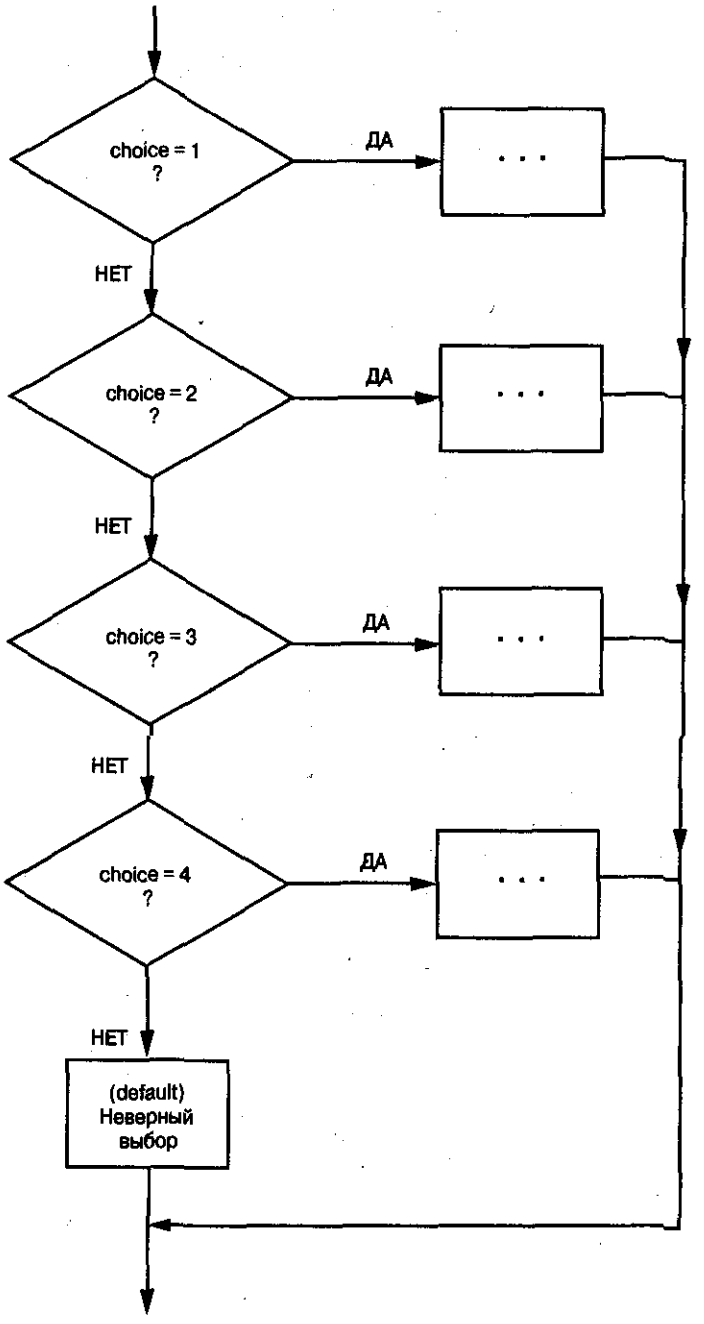


Рис. 3.14. Блок-схема, поясняющая действие оператора `switch`

Прерывание потока управления

Несмотря на то что разработчики языка Java сохранили зарезервированное слово goto, они решили не включать его в язык. В принципе применение операторов goto считается признаком плохого стиля программирования. Некоторые программисты считают, что борьба с использованием оператора goto ведется недостаточно активно (см., например, известную статью Дональда Кнута (Donald Knuth) "Structured Programming with goto statements"). Они считают, что применение операторов goto может приводить к ошибкам, однако в некоторых случаях нужно выполнять преждевременный *выход из цикла*. Создатели языка Java согласились с их аргументами и даже добавили в язык новый оператор для поддержки такого стиля программирования — оператор break с меткой.

Рассмотрим обычный оператор break, в котором не применяется метка. Для выхода из цикла можно применять тот же оператор, который использовался для выхода из тела оператора switch. Пример использования оператора break приведен ниже.

```
while (years <= 100)
{
   balance += payment;
   double interest = balance * interestRate / 100;
   balance += interest;
   if(balance >= goal) break;
   years++;
}
```

Теперь выход из цикла осуществляется при выполнении одного из двух условий: years > 100 — в начале цикла либо balance >= goal — внутри цикла. Разумеется, те же действия можно было бы вычислить и без применения оператора break.

```
while (years <= 100 && balance < goal)
{
   balance += payment;
   double interest = balance * interestRate / 100;
   balance += interest;
   if(balance < goal)
   years++;
}
```

Заметим, однако, что проверка условия balance < goal в данном варианте программы повторяется дважды. Оператор break позволяет избежать этого.

В отличие от языка C++, язык Java содержит также оператор break *с меткой*, обеспечивающий выход из вложенных циклов. Если во вложенном цикле выполняется некоторое условие, возможно, потребуется выйти из всех вложенных циклов. Программировать дополнительные условия для проверки каждого вложенного цикла попросту неудобно.

Ниже приводится пример, иллюстрирующий работу оператора break с меткой. Заметим, что метка должна предшествовать тому внешнему циклу, из которого требуется выйти. Метка должна оканчиваться двоеточием.

```
Scanner in = new Scanner(System.in);
int n;
read_data:
while (. . .) // Этот цикл помечен
{
```

```
   . . .
   for (. . .) // Этот цикл не помечен
   {
      System.out.print("Введите число >= 0: ");
      n = in.nextInt();
      if (n < 0) // Если это не произойдет, цикл продолжится
         break read_data;
         // Прерывание цикла
      . . .
   }
}
// Данное выражение выполняется сразу же после оператора break с меткой
if (n < 0) // Проверка на наличие недопустимой ситуации
{
   // Обработка недопустимой ситуации
}
else
{
   // Действия в результате нормального выполнения программы
}
```

Если в программу было введено неверное число, оператор `break` с меткой выполнит переход в конец помеченного блока. В этом случае необходимо проверить, нормально ли осуществлен выход из цикла, или он произошел в результате выполнения оператора `break`.

Это может показаться странным, но метку можно связать с любым оператором, даже оператором `if` или блоком. Соответствующий пример приведен ниже.

```
метка:
{
   ...
   if (условие) break label; // Выход из блока.
   ...
}
// При выполнении оператора break управление передается в эту точку.
```

Итак, если вам крайне нужен оператор `goto`, вы можете поместить блок, из которого нужно выйти, и применить оператор `break`! Естественно, мы не рекомендуем использовать этот способ. Заметим также, что подобным образом можно выйти из блока, но невозможно войти в него.

Существует также оператор `continue`, который, подобно `break`, прерывает нормальное выполнение программы. Оператор `continue` передает управление в начало текущего вложенного цикла. Пример использования данного оператора приведен ниже.

```
while (sum < goal)
{
   String input = JOptionPane.showInputDialog("Введите число");
   n = integer.ParseInt(input);
   if (n < 0) continue;
   sum += n; // Не выполняется, если n < 0.
}
```

Если n < 0, то оператор continue выполняет переход в начало цикла, пропуская оставшуюся часть текущей итерации.

Если оператор continue используется в цикле for, он передает управление оператору увеличения счетчика цикла.

```
for (count = 0; count < 100; count ++)
{
   String input = JOptionPane.showInputDialog("Ввод числа");
   n = Integer.parseInt(input);
   if (n < 0) continue;
   sum += n; // Не выполняется, если n < 0.
}
```

Если n < 0, то оператор continue выполнит переход к оператору count++.

В языке Java есть также оператор continue с меткой, передающий управление заголовку оператора цикла, помеченного соответствующей меткой.

Многие программисты считают, что операторы break и continue неоправданно усложняют текст программы. Применять эти операторы вовсе не обязательно — те же действия можно реализовать, не прибегая к ним. В этой книге мы нигде не используем ни break, ни continue.

Работа с большими числами

Если для решения задачи недостаточно точности встроенных целочисленных типов и чисел с плавающей точкой, можно обратиться к классам BigInteger и BigDecimal, принадлежащим пакету java.math. Эти классы предназначены для выполнения действий с числами, состоящими из произвольного количества цифр. Классы BigInteger и BigDecimal реализуют арифметические операции произвольной точности соответственно для целых и действительных чисел.

Для преобразования обычного числа в число с произвольной точностью (называемое также большим числом) используется статический метод valueOf():

```
BigInteger a = BigInteger.valueof(100);
```

К сожалению, к большим числам нельзя применять обычные математические операторы, например + или *. Вместо этого надо использовать методы add() и multiply() из соответствующих классов.

```
BigInteger c = a.add(b); // c = a + b
BigInteger d = c.multiply(b.add(BigInteger.value.Of(2)));
   // d = c * (b + 2)
```

В отличие от языка C++, язык Java не поддерживает перегрузку операторов. Поэтому разработчики класса BigInteger были лишены возможности переопределить операторы + и * для методов add() и multiply() в классе BigInteger.

В листинге 3.6 показана модифицированная программа для подсчета шансов выиграть в лотерее (ее исходный вариант см. в листинге 3.5). Теперь эта программа может работать с большими числами. Например, если вам предложили сыграть в лотерее, в которой нужно угадать 60 чисел из 490 возможных, то эта программа сообщит вам, что шанс выиграть равен 1 из 7163958434619955574151162254009293341717612789263493493351013459481104668848. Удачи!

Программа, представленная в листинге 3.5, вычисляла следующее выражение:

```
lotteryOdds = lottery * (n - i + 1) / i;
```

При работе с большими числами соответствующая строка кода выглядит так:

```
lotteryOdds = lotteryOdds.multiply(BigInteger.valueOf(n-i+1)
              .divide(BigInteger.valueOf(i));
```

Листинг 3.6. Содержимое файла `BigIntegerTest.java`

```
import java.math.*;
import java.util.*;

public class BigIntegerTest
{
   public static void main(String[] args)
   {
      Scanner in = Scanner.create(System.in);

      System.out.print(
           "Сколько номеров? ");
      int k = in.nextInt();

      System.out.print(
         "Наибольший номер? ");
      int n = in.nextInt();

      /*
         Вычисление выражения
         n * (n - 1) * (n - 2) * . . . * (n - k + 1)
         ------------------------------------------
         1 * 2 * 3 * . . . * k
      */

      BigInteger lotteryOdds = BigInteger.valueOf(1);

      for (int i = 1; i <= k; i++)
         lotteryOdds = lotteryOdds
            .multiply(BigInteger.valueOf(n - i + 1))
            .divide(BigInteger.valueOf(i));

      System.out.println("Ваши шансы 1 из " + lotteryOdds +
        ". Успехов!");
   }
}
```

API **java.math.BigInteger 1.1**

- `BigInteger add(BigInteger other)`
- `BigInteger subtract(BigInteger other)`
- `BigInteger multiply(BigInteger other)`
- `BigInteger divide(BigInteger other)`
- `BigInteger mod(BigInteger other)`

Возвращают сумму, разность, произведение, частное и остаток от деления, полученные в результате выполнения соответствующих операций над данным большим числом и значением переменной other.

- `int compareTo(BigInteger other)`

  Возвращает 0, если данное большое число равно значению переменной other, отрицательное число, если данное большое число меньше значения переменной other, и положительное число в противном случае.

- `static BigInteger valueOf(long x)`

  Возвращает большое число, значение которого равно значению переменной x.

java.math.BigDecimal 1.1

- `BigDecimal add(BigDecimal other)`
- `BigDecimal subtract(BigDecimal other)`
- `BigDecimal multiply(BigDecimal other)`
- `BigDecimal divide(BigDecimal other, int roundingMode) 5.0`

  Возвращает сумму, разность, произведение и частное, полученные в результате выполнения соответствующих операций над данным большим числом и значением переменной other. Чтобы вычислить частное, нужно указать *режим округления*. Режим roundingMode.ROUND_HALF_UP означает привычное всем округление (т.е. в сторону уменьшения для цифры меньше 5 и в сторону увеличения, если соответствующая цифра больше или равна 5). Для обычных вычислений этого достаточно. Другие режимы округления описаны в документации.

- `int compareTo(BigDecimal other)`

  Возвращает 0, если данное число типа BigDecimal равно значению переменной other, отрицательное число, если данное число меньше значения переменной other, и положительное число в противном случае.

- `static BigDecimal valueOf(long x)`
- `static BigDecimal valueOf(long x, int scale)`

  Возвращает большое десятичное число, значение которого равно значению переменной x или $x/10^{scale}$.

Массивы

Массив — это структура данных, в которой хранятся величины одинакового типа. Доступ к отдельному элементу массива осуществляется с помощью целочисленного *индекса*. Например, если a — массив целых чисел, то значение выражения a[i] равно i-му целому числу в массиве.

Массив объявляется следующим образом: сначала указывается тип массива, т.е тип элементов, содержащихся нем, затем следует пара пустых квадратных скобок, а после них — имя переменной. Ниже приведено объявление массива, состоящего из целых чисел.

```
int[] a;
```

Однако этот оператор лишь объявляет переменную a, не инициализируя ее. Чтобы создать массив, нужно применить оператор new.

```
int[] a = new int[100];
```

Этот оператор создает массив, состоящий из 100 целых чисел.

Объявить массив можно двумя способами:

```
int[] a;
```

или

```
int a[];
```

Большинство программистов, работающих на Java, используют первый способ, так как при этом тип более явно отделяется от имени переменной.

Элементы сформированного выше массива *нумеруются от 0 до 99* (а не от 1 до 100). После создания массив можно заполнять конкретными значениями, в частности, это можно делать в цикле.

```
int[] a = new int[100];
for (int i = 0; i < 100; i++)
   a[i] = i; // Заполняет массив числами от 0 до 99.
```

Если, создав массив, состоящий из 100 элементов, вы попытаетесь обратиться к a[100] (или любому другому элементу, индекс которого выходит за пределы диапазона от 0 до 99), программа прекратит работу, поскольку будет сгенерировано исключение, соответствующее выходу индекса массива за пределы допустимого диапазона.

Чтобы подсчитать количество элементов в массиве, используйте выражение *имя_массива*.length. Например:

```
for (int i = 0; i < a.length; i++)
   System.out.println(a[i]);
```

После создания массива изменить его размер невозможно (хотя можно, конечно, изменять его отдельные элементы). Если в ходе выполнения программы необходимо часто изменять размер массива, лучше использовать другую структуру данных, называемую *списком*.

Цикл "for each"

В JDK 5.0 был реализован новый цикл, позволяющий перебирать все элементы массива (а также любого другого набора данных), не применяя счетчик.

Новый вариант цикла for записывается следующим образом:

```
for (переменная : набор_данных) выражение
```

При обработке цикла переменной последовательно присваивается каждый элемент набора данных, после чего выполняется выражение (или блок). В качестве набора данных может использоваться массив либо экземпляр класса, реализующего интерфейс Iterable, например ArrayList. Списки будут обсуждаться в главе 5, а интерфейс Iterable — в главе 2 второго тома.

Ниже приведен пример использования рассматриваемого здесь цикла.

```
for (int element : a)
   System.out.println(element);
```

В результате выполнения данного фрагмента кода каждый элемент массива будет выведен в отдельной строке.

Действия данного цикла можно кратко описать как "обработка каждого элемента из a". Разработчики языка Java рассматривали возможность применения в качестве идентификатора данного цикла ключевых слов `foreach` и `in`. Однако данный тип цикла появился намного позже основных языковых средств Java, и введение нового ключевого слова привело бы к необходимости изменять исходный код некоторых готовых приложений, содержащих переменные или методы с такими именами.

Следует заметить, что действия, выполняемые с помощью нового цикла, можно реализовать с помощью традиционного цикла `for`.

```
for (int i = 0; i < a.length; i++)
   System.out.println(a[i]);
```

Однако при использовании нового цикла запись получается более краткой (необходимость в начальном выражении и условии завершения цикла отпадает), и вероятность возникновения ошибок уменьшается.

Переменная цикла в выражении "for each" перебирает не значения индекса, а элементы массива.

Несмотря на появление нового цикла "for each", упрощающего во многих случаях составление программ, традиционный цикл `for` отнюдь не устарел. Без него нельзя обойтись, например, в тех случаях, когда вам надо обработать не весь набор данных, а лишь его часть, либо тогда, когда счетчик явно используется в теле цикла.

Инициализация массивов и анонимные массивы

В языке Java есть средство для одновременного создания массива и его инициализации. Пример такой синтаксической конструкции приведен ниже.

```
int[] smallPrimes = { 2, 3, 5, 7, 11, 13};
```

Заметьте, что в этом случае оператор `new` не нужен.

Кроме того, можно даже инициализировать массив, не имеющий имени, или *анонимный массив*.

```
new int[] {16, 19, 23, 29, 31, 37}
```

Данное выражение выделяет память для нового массива и заполняет его числами, указанными в фигурных скобках. При этом подсчитывается их количество и соответственно определяется размер массива. Такую синтаксическую конструкцию удобно применять для повторной инициализации массива без образования новой переменной. Например, выражение

```
smallPrimes = new int{ 17, 19, 23, 29, 31, 37 };
```

представляет собой сокращенную запись выражения

```
int[] anonymous = { 17, 19, 23, 29, 31, 37 };
smallPrimes = anonymous;
```

При необходимости можно создать массив нулевого размера. Такой массив может оказаться полезным при написании метода, возвращающего массив, который в некоторых случаях оказывается пустым. Массив нулевой длины объявляется следующим образом:

```
new тип_элементов[0]
```

Заметим, что такой массив не эквивалентен объекту `null`. (Подробно этот вопрос будет обсуждаться в главе 4.)

Копирование массивов

По мере необходимости одну переменную массива можно скопировать в другую, но при этом *обе переменные будут ссылаться на один и тот же массив.*

```
int[] luckyNumbers = smallPrimes;
luckyNuimbers[5] = 12; // Теперь элемент smallPrimes[5]также равен 12.
```

Результат копирования переменных массивов показан на рис. 3.15. Если необходимо скопировать все элементы одного массива в другой, следует использовать метод `arraycopy()` из класса `System`. Его вызов выглядит следующим образом:

```
System.arraycopy(from, fromIndex, to, toIndex, count);
```

Массив `to` должен иметь достаточный размер, чтобы в нем поместились все копируемые элементы.

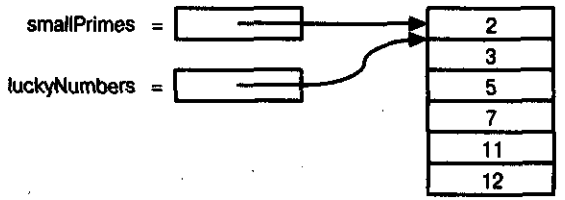


Рис. 3.15. Копирование переменной массива

Ниже приведен фрагмент кода, результаты выполнения которого показаны на рис. 3.16. Вначале создаются два массива, а затем последние четыре элемента первого массива копируются во второй. Копирование исходного массива начинается с элемента с номером 2, а в целевой массив копируемые данные помещаются, начиная с элемента с номером 3.

```
int[] smallPrimes = {2, 3, 5, 7, 11, 13};
int[] luckyNumbers = {1001, 1002, 1003, 1004, 1005, 1006, 1007};
System.arraycopy(smallPrimes, 2, luckyNumbers, 3, 4);
for ( int i = 0; i < luckyNumbers.length; i++)
   System.println(i + ":  " + luckyNumbers[i]);
```

Выполнение данного фрагмента приводит к следующему результату:

```
0:  1001
1:  1002
2:  1003
3:  5
4:  7
5:  11
6:  13
```

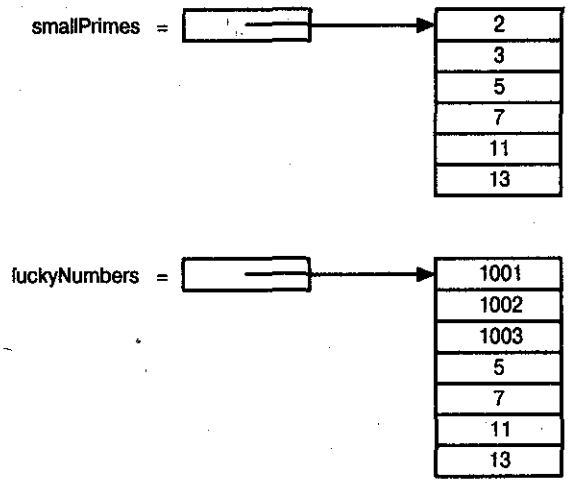


Рис. 3.16. Копирование элементов массива

Массив в языке Java значительно отличается от массива в языке C++. Однако переменную массива можно условно сравнить с указателем на *динамически созданный* массив. Таким образом, выражение Java

```
int[] a = new int[100];  //Java
```

можно сравнить с выражением C++

```
int* a = new int[100]; // C++
```

но оно существенно отличается от следующего:

```
int a[100]; // C++
```

В языке Java оператор `[]` по умолчанию проверяет *диапазон изменения индексов*. Кроме того, в языке Java не поддерживается арифметика указателей, — нельзя увеличить указатель `a`, чтобы обратиться к следующему элементу массива.

Параметры командной строки

В каждой из рассмотренных ранее программ на языке Java присутствовал метод `main()` с параметром `String[] args`. Этот параметр означает, что метод `main()` получает массив, элементами которого являются параметры, указанные в командной строке.

Рассмотрим следующую программу:

```
public class Message
{
   public static void main(String[] args)
   {
      if (args[0].equals("-h");
         System.out.print("Hello, ");
      else if (args[0].equals("-g"))
         System.out.print("Goodbye, ");
      // Вывод остальных параметров командной строки.
      for (int i = 1; i < args.length; i++)
```

```
            System.out.print(" " + args[i]);
         System.out.print("!");
      }
}
```

При следующем вызове программы

```
java Message -g cruel world
```

массив `args` будет состоять из таких элементов:

```
args[0]: "-g"
args[1]: "cruel"
args[2]: "world"
```

Программа выведет сообщение

```
Goodbye, cruel world!
```

При запуске программы на языке Java ее имя не помещается в массиве `args`. Например, *после запуска из командной строки программы* `Message` *с помощью команды*

```
java Message -h world
```

элемент `args[0]` будет равен `"-h"`, а не `"Message"` или `"java"`.

Сортировка массива

Если нужно упорядочить массив чисел, можно применить метод `sort()` из класса `Arrays`.

```
int[] a = new int[10000];
...
Arrays.sort(a);
```

Этот метод использует усовершенствованную версию алгоритма быстрой сортировки, которая считается наиболее эффективной для большинства множеств. Класс `Arrays` содержит несколько удобных методов, предназначенных для работы с массивами. Эти методы приведены в конце данного раздела.

Программа, код которой представлен в листинге 3.7, создает массив и генерирует случайную комбинацию чисел для лотереи. Например, если нужно выиграть "6 из 49", программа может вывести следующее сообщение:

```
Попробуйте такую комбинацию, и станете богатым!
   4
   7
   8
   19
   30
   44
```

Чтобы выбрать случайные числа, массив `numbers` сначала заполняется числами 1, 2, ..., n.

```
int[] numbers = new int[n];
for (int i = 0; i < numbers.length; i++)
   numbers[i] = i + 1;
```

Второй массив предназначен для хранения сгенерированных чисел.

```
int[] result = new int[k];
```

Теперь сгенерируем k чисел. Метод `Math.random()` возвращает случайное число с плавающей точкой, лежащее в интервале от 0 (включительно) до 1 (это значение не принадлежит интервалу). Умножая результат на число n, получим случайное число, лежащее между 0 и n-1.

```
int r = (int)(Math.random() * n);
```

Присвоим i-е число i-му элементу массива. Сначала там будет помещено само число r, однако, как будет показано ниже, содержимое массива number будет изменяться после генерации каждого нового числа.

```
result[i] = numbers[r];
```

Теперь мы должны убедиться, что ни одно число не повторится — все номера должны быть разными. Следовательно, нужно поместить в элемент `number[r]` *последнее* число, содержащееся в массиве, и уменьшить n на единицу.

```
numbers[r] = numbers[n - 1];
n--;
```

Обратите внимание на то, что при каждой генерации мы получаем *индекс*, а не само число. Этот индекс относится к массиву, содержащему числа, которые еще не были выбраны.

После генерации k номеров полученный массив сортируется, чтобы результат выглядел более элегантно.

```
Arrays.sort(result);
for (int i = 0; i < result.length; i++)
   System.out.println(result[i]);
```

Листинг 3.7. Содержимое файла `LotteryDrawing.java`

```
import java.util.*;

public class LotteryDrawing
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      System.out.print("Число номеров? ");
      int k = in.nextInt();

      System.out.print("Максимальный номер? ");
      int n = in.nextInt();

      // Заполнение массива числами 1 2 3 . . . n
      int[] numbers = new int[n];
      for (int i = 0; i < numbers.length; i++)
         numbers[i] = i + 1;

      // Выбор k номеров и запись их в другой массив
      int[] result = new int[k];
```

```
      for (int i = 0; i < result.length; i++)
      {
         // Генерация случайного индекса между 0 и n - 1
         int r = (int) (Math.random() * n);

         // Выбор случайного элемента
         result[i] = numbers[r];

         // Копирование последнего элемента
         numbers[r] = numbers[n - 1];
         n--;
      }

      // Вывод сортированного массива
      Arrays.sort(result);
      System.out.println(
      "Попробуйте такую комбинацию, и станете богатым!");
      for (int r : result)
         System.out.println(r);
   }
}
```

API java.lang.System 1.1

- `static void arraycopy(Object from, int fromIndex, Object to, int toIndex, int count)`

  Копирует элементы из первого массива во второй.

Параметры		
	`from`	Массив любого типа. (Причины, по которым этот параметр должен иметь тип `Object`, объясняются в главе 5.)
	`fromIndex`	Индекс, с которого начинается копирование элементов исходного массива
	`to`	Массив, имеющий тот же тип, что и массив `from`
	`toIndex`	Индекс, с которого начинается копирование элементов в целевой массив
	`count`	Количество копируемых элементов

API java.util.Arrays 1.2

- `static void sort(mun[] a)`

  Упорядочивает массив с помощью алгоритма быстрой сортировки.

Параметр		
	`a`	Массив типа `int`, `long`, `short`, `char`, `byte`, `boolean`, `float` или `double`

- `static int binarySearch(mun[] a, mun v)`

  Метод использует алгоритм бинарного поиска величины `v`. Если требуемый элемент найден, возвращается его индекс. В противном случае возвращается

отрицательное значение r; значение v должно быть вставлено в ячейку -r-1 массива a.

Параметры	a	Упорядоченный массив типа int, long, short, char, byte, boolean, float или double
	v	Значение того же типа, что и элементы массива a

- `static void fill(`*mun*`[] a, `*mun*` v)`

  Присваивает каждому элементу массива значение v.

Параметры	a	Массив типа int, long, short, char, byte, boolean, float или double
	v	Значение того же типа, что и элементы массива a

- `static boolean equals(`*mun*`[] a, `*mun*`[] b)`

  Возвращает величину true, если массивы имеют одинаковую длину и элементы с соответствующими индексами совпадают.

Параметры	a, b	Массив типа int, long, short, char, byte, boolean, float или double

Многомерные массивы

Для доступа к элементам многомерного массива применяется несколько индексов. Такие массивы используются для хранения таблиц и более сложных упорядоченных структур данных. Если многомерные массивы в вашей работе не нужны, можете смело пропустить этот раздел.

Допустим, что вам нужно создать таблицу чисел, показывающих, как возрастет первоначальная инвестиция объемом 10000 долларов при разных процентных ставках, если прибыль ежегодно выплачивается и реинвестируется. Необходимые данные показаны в табл. 3.8.

Таблица 3.8. Рост вклада при разных процентных ставках

10%	11%	12%	13%	14%	15%
10000,00	10000,00	10000,00	10000,00	10000,00	10000,00
11000,00	11100,00	11200,00	11300,00	11400,00	11500,00
12100,00	12321,00	12544,00	12769,00	12996,00	13225,00
13310,00	13676,31	14049,28	14428,97	14815,44	15208,75
14641,00	15180,70	15735,19	16304,74	16889,60	17490,06
16105,10	16850,58	17623,42	18424,35	19254,15	20113,57
17715,61	18704,15	19738,23	20819,52	21949,73	23130,61
19487,17	20761,60	22106,81	23526,05	25022,69	26600,20
21435,89	23045,38	24759,63	26584,44	28525,86	30590,23
23579,48	25580,37	26730,79	30040,42	32519,49	35178,76

Очевидно, что эту информацию лучше всего хранить в двухмерном массиве (или матрице), который мы назовем balance.

Объявить двухмерный массив в языке Java довольно просто. Например, это можно сделать следующим образом:

```
double[][] balances;
```

Как обычно, мы не можем использовать массив, пока он не инициализирован с помощью оператора new.

```
balances = new double[NYEARS][NRATES];
```

В других случаях, если элементы массива известны заранее, можно использовать сокращенную запись для его инициализации, в которой не применяется оператор new.

```
int[][] magicSquare =
   {
     {16, 3, 2, 13},
     {5, 10, 11, 8},
     {9, 6, 7, 12},
     {4, 15, 14, 1}
   };
```

После инициализации массива к его отдельным элементам можно обращаться с помощью двух пар квадратных скобок, например `balances[i][j]`.

Программа, приведенная в качестве примера, сохраняет процентные ставки в одномерном массиве `interest`, а результаты подсчета баланса для каждого года и каждой процентной ставки — в двухмерном массиве `balance`. Первая строка массива инициализируется исходной суммой.

```
for (int j = 0; j < balances[0].length; j++)
   balances[0][j] = 10000;
```

Затем мы подсчитываем содержимое остальных строк.

```
for (int i = 1; i < balances.length; i++)
{
   for (int j = 0; j < balances[i].length; j++)
   {
      double oldBalance = balances[i - 1][j];
      double interest = . . .;
      balances[i][j] = oldBalance + interest;
   }
}
```

В листинге 3.8 код программы показан полностью.

Цикл "for each" не обеспечивает автоматического перебора элементов двухмерного массива. Он лишь перебирает строки, которые, в свою очередь, являются одномерными массивами. Для обработки всех элементов двухмерного массива нужно два цикла.

```
for (double[] row : balances)
   for (double b : row)
      Обработка b
```

Листинг 3.8. Содержимое файла CompoundInterest.java

```
public class CompoundInterest
{
   public static void main(String[] args)
   {
      final int STARTRATE = 10;
      final int NRATES = 6;
      final int NYEARS = 10;

      // Установка процентной ставки 10 . . . 15%.
      double[] interestRate = new double[NRATES];
      for (int j = 0; j < interestRate.length; j++)
         interestRate[j] = (STARTRATE + j) / 100.0;

      double[][] balances = new double[NYEARS][NRATES];

      // Исходная сумма равна 10000
      for (int j = 0; j < balances[0].length; j++)
         balances[0][j] = 10000;

      // Вычисление баланса на следующие годы.
      for (int i = 1; i < balances.length; i++)
      {
         for (int j = 0; j < balances[i].length; j++)
         {
            // Извлечение баланса за прошлый год.
            double oldBalance = balances[i - 1][j];

            // Вычисление процентного дохода.
            double interest = oldBalance * interestRate[j];

            // Вычисление баланса за текущий год.
            balances[i][j] = oldBalance + interest;
         }
      }

      // Вывод процентных ставок.
      for (int j = 0; j < interestRate.length; j++)
         System.out.printf("%9.0f%%", 100 * interestRate[j]);

      System.out.println();

      // Вывод таблицы.
      for (double[] row : balances)
      {
         // Вывод строки таблицы.
         for (double b : row)
            System.out.printf("%10.2f", b);

         System.out.println();
      }
   }
}
```

"Неровные" массивы

Все языковые конструкции, которые мы до сих пор рассматривали, мало отличались от других языков программирования. Однако механизм массивов в Java имеет особенность, предоставляющую совершенно новые возможности. В данном языке вообще *нет* многомерных массивов, только одномерные. Многомерные массивы — это "массивы массивов".

Например, массив `balances` в предыдущем примере фактически представляет собой массив, состоящий из 10 элементов, каждый из которых является массивом из шести элементов, представляющих собой числа с плавающей точкой (рис. 3.17).

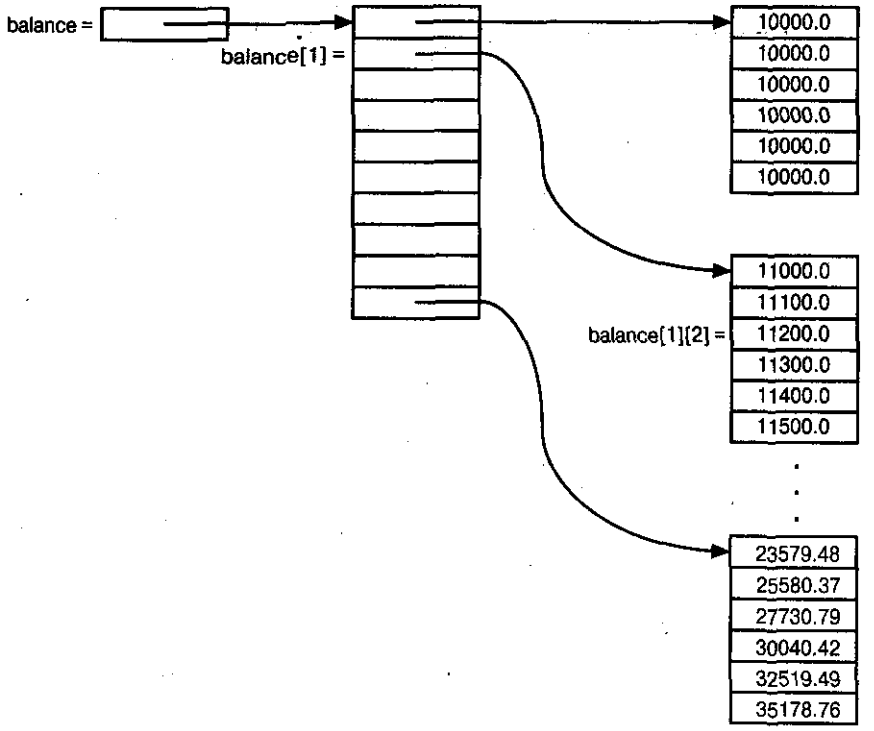


Рис. 3.17. Двухмерный массив

Выражение `balance[i]` определяет i-й подмассив, т.е i-ю строку таблицы. Эта строка сама представляет собой массив, и выражение `balance[i][j]` относится к его j-му элементу.

Поскольку строки массива доступны из программмы, их можно легко переставлять!

```
double[] temp = balances[i];
balances[i] = balances[i+1];
balances[i=1] = temp;
```

Кроме того, в языке Java легко создавать "неровные" массивы, т.е массивы, у которых разные строки имеют разную длину. Приведем стандартный пример. Создадим

массив, в котором элемент, стоящий на пересечении i-й строки и j-го столбца, равен количеству возможностей "выбрать j чисел из i".

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5  10  10   5   1
1   6  15  20  15   6   1
```

Поскольку число j не может превышать i, получается треугольная матрица. В i-й строке этой матрицы находится i+1 элементов. (Мы можем выбрать и 0 элементов; сделать это можно лишь одним-единственным способом.) Чтобы создать неровный массив, сначала разместим в памяти массив, хранящий его строки.

```
int[][] odds = new int[NMAX+1][];
```

Затем поместим туда сами строки.

```
for (int n=0; n<=NMAX; n++)
   odds[n] = new int[n+1];
```

Теперь в памяти размещен весь массив, так что мы можем обращаться к его элементам, как обычно, при условии, что индексы не выходят за пределы допустимого диапазона.

```
for (int n=0; n<odds.length; n++)
   for (int k=0; k<odds[n].length; k++)
   {
      // Вычисление вариантов.
      ...
      odds[n][k] = lotteryOdds;
   }
```

В листинге 3.9 приведен полный текст данной программы.

В языке Java объявление

```
double[][] balance = new double[10[6]; // Java
```

не эквивалентно объявлению

```
double balance[10][6]; // C++
```

и даже не соответствует

```
double (*balance)[6] = new double[10][6]; // C++
```

Вместо этого в памяти размещается массив, состоящий из десяти указателей. Средствами C++ это можно выразить следующим образом:

```
double** balance = new double*[10]; // C++
```

Затем каждый элемент в массиве указателей заполняется массивом, состоящим из 6 чисел.

```
for (i = 0; i < 10; i++)
  balance[i] = new double[6];
```

К счастью, этот цикл выполняется автоматически при объявлении массива с помощью оператора `new double[10][6]`. Если вам нужны "неровные" массивы, размещайте массивы строк отдельно.

Листинг 3.9. Содержимое файла LotteryArray.java

```
public class LotteryArray
{
   public static void main(String[] args)
   {
      final int NMAX = 10;

      // Выделение памяти для треугольной матрицы
      int[][] odds = new int[NMAX + 1][];
      for (int n = 0; n <= NMAX; n++)
         odds[n] = new int[n + 1];

      // Заполнение треугольной матрицы
      for (int n = 0; n < odds.length; n++)
         for (int k = 0; k < odds[n].length; k++)
         {
            /*
               Вычисление выражения
               n * (n - 1) * (n - 2) * . . . * (n - k + 1)
               ------------------------------------------
               1 * 2 * 3 * . . . * k
            */
            int lotteryOdds = 1;
            for (int i = 1; i <= k; i++)
               lotteryOdds = lotteryOdds * (n - i + 1) / i;

            odds[n][k] = lotteryOdds;
         }

      // Вывод треугольной матрицы
      for (int[] row : odds)
      {
         for (int odd : row)
            System.out.printf("%4d", odd);
         System.out.println();
      }
   }
}
```

Глава 4

Объекты и классы

В этой главе...

- Введение в объектно-ориентированное программирование
- Использование готовых классов
- Определение собственных классов
- Статические поля и методы
- Параметры методов
- Формирование объектов
- Пакеты
- Комментарии и документирование
- Рекомендации по разработке классов

В этой главе рассматриваются следующие вопросы.

- Введение в объектно-ориентированное программирование.
- Создание объектов, соответствующих классам из стандартной библиотеки Java.
- Создание собственных классов.

Если вы недостаточно хорошо ориентируетесь в вопросах объектно-ориентированного программирования, внимательно прочитайте данную главу. Для создания объектов требуется совершенно иной способ мышления по сравнению подходом, типичным для процедурно-ориентированных языков. Освоить новые принципы создания программ не всегда просто, но сделать это необходимо; для овладения языком Java надо хорошо знать основные понятия объектно-ориентированного программирования.

Программистам, имеющим большой опыт работы с языком C++, информация, содержащаяся в этой и в предыдущей главах, уже известна. Однако между Java и C++ есть существенные различия, поэтому последний раздел данной главы следует прочесть очень внимательно.

Введение в объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) в настоящее время стало доминирующей парадигмой программирования, вытеснив "структурные", процедурно-ориентированные подходы, разработанные в начале 1970-х годов. Java представля-

ет собой полностью объектный язык, и на нем невозможно создавать привычные процедурные программы. Мы надеемся, что, прочитав эту главу и проанализировав примеры, приведенные в тексте, вы сможете продуктивно работать на языке Java.

Начнем с вопроса, который на первый взгляд не относится к программированию: каким образом такие компании, как Compaq, Dell, Gateway и другие производители персональных компьютеров, так быстро стали крупными фирмами? Возможно, большинство читателей ответят, что они делали хорошее оборудование и продавали его по очень низким ценам, к тому же в то время спрос на компьютеры был очень высоким. Поставим вопрос несколько иначе: как они смогли в короткий срок произвести так много моделей, реагируя на постоянно изменяющиеся требования?

В основном причина успеха заключается в том, что компании поручили бо́льшую часть работы другим компаниям. Они покупали компоненты у поставщиков с хорошей репутацией, а затем собирали их в единое целое на своих заводах. Производители компьютеров не тратили время и деньги на разработку и создание блоков питания, дисководов, материнских плат и других компонентов. Они давали возможность другим компаниям производить свою продукцию и быстро реагировать на изменения спроса. Если бы производители компьютеров разрабатывали все блоки самостоятельно, они бы затрачивали на выпуск каждой модели неизмеримо больше усилий.

Производители компьютеров, приобретая изделия, относились к ним как к "черным ящикам". Например, покупая блоки питания, они приобретали нечто, обладающее определенными свойствами (размером, формой и т.д.) и способное выполнять определенные функции (сглаживать выходные сигналы, обеспечивать необходимую мощность и т.д.). Достаточно рассмотреть деятельность компании Compaq, чтобы убедиться в высокой эффективности такого подхода. Перейдя от собственного производства компонентов компьютера к их покупке и последующей сборке, фирма резко улучшила свое финансовое положение.

В основе ООП лежит та же идея. Любая программа состоит из объектов, обладающих определенными свойствами, в частности способными выполнять определенные операции. Создавать ли объект самостоятельно или покупать его у других программистов — решение зависит лишь от наличия денег и времени. Если объекты удовлетворяют вашим требованиям, неважно, как именно они сделаны. В ООП нужно заботиться лишь о том, *что представляет собой* объект. Производители компьютеров не вникают в детали внутреннего устройства блоков питания, пока эти блоки работают корректно. Точно так же большинство программистов, работающих на языке Java, не интересуются, как именно реализован тот или иной объект; главное, чтобы он удовлетворял их требованиям.

Традиционное структурное программирование заключается в разработке набора процедур (или *алгоритмов*) для решения поставленной задачи. Определив эти процедуры, программист должен задать подходящий способ хранения данных. Вот почему создатель языка Pascal Никлаус Вирт (Niklaus Wirth) назвал свою знаменитую книгу по программированию *Алгоритмы + Структуры данных = Программы*. Заметьте, что в названии этой книги алгоритмы стоят на первом месте, а структуры данных — на втором. Это отражает образ мышления программистов того времени. Во-первых, они решали, как манипулировать данными; затем решали, какую структуру применить для организации этих данных, чтобы работать с ними было легче. Объектный подход в корне изменил ситуацию, поместив на первое место данные и лишь на второе — алгоритмы, предназначенные для их обработки.

Основной принцип, обеспечивший высокую эффективность ООП, гласит: каждый объект предназначен для выполнения определенных задач. Если перед объектом стоит задача, для решения которой он не предназначен, у него должен быть доступ к другому объекту, который может решить эту задачу. Другими словами, первый объект просит второй решить поставленную задачу. Это – обобщенный вариант обращений, применяемых в процедурном программировании. (Напомним, что в языке Java это обычно называется *вызовом метода*.)

Объект никогда не должен непосредственно манипулировать внутренними данными другого объекта, а также предоставлять другим объектам непосредственный доступ к своим данным. Все связи между объектами обеспечиваются с помощью вызовов методов. *Инкапсуляция* (encapsulation) данных объекта максимально повышает возможность его повторного использования, уменьшает взаимозависимость объектов и минимизирует время отладки программы.

Разумеется, от объектов не следует ожидать слишком многого. Этим они схожи с модулями, применяемыми в процедурно-ориентированном программировании. И разработка, и отладка программы, состоящей из небольших объектов, выполняющих частные задачи, намного проще по сравнению с программой, созданной из громадных объектов с крайне сложными внутренними данными и сотнями функций для манипулирования ими.

Основные термины объектно-ориентированного программирования

Для дальнейшей работы вам нужно освоить терминологию ООП. Наиболее важным термином является *класс*, который мы уже видели в примерах программ из главы 3. Класс – это шаблон, или проект, по которому будет сделан объект. Обычно класс сравнивают с формой для выпечки печенья. Объект – это само печенье. *Конструирование* объекта на основе некоторого класса называется созданием *экземпляра* (instance) этого класса.

Как мы уже видели, все коды, которые создаются в языке Java, находятся внутри классов. Стандартная библиотека языка Java содержит несколько тысяч классов, предназначенных для решения разных задач, например, для создания пользовательского интерфейса, календарей, установления сетевых соединений и т.д. Несмотря на это, программисты продолжают создавать свои собственные классы на языке Java, чтобы формировать объекты, характерные для разрабатываемого приложения, а также адаптировать классы из стандартной библиотеки для своих нужд.

Инкапсуляция (иногда называемая *сокрытием данных*) – это ключевое понятие при работе с объектами. Формально инкапсуляцией считается обычное объединение данных и операций над ними в одном пакете и сокрытие данных от других объектов. Данные в объекте называются *полями экземпляра* (instance fields), а функции и процедуры, выполняющие операции над данными, – его *методами* (methods). В конкретном объекте, т.е. экземпляре класса, поля экземпляра имеют определенные значения. *Множество этих значений называется текущим состоянием* (state) *объекта*. Применение любого метода к какому-нибудь объекту может изменить его состояние.

Еще раз подчеркнем, что основной принцип инкапсуляции заключается в запрещении прямого доступа к полям экземпляра данного класса из других классов. Программы должны взаимодействовать с данными объекта *только* с помощью методов этого объекта. Из сказанного следует, что в классе можно полностью изменить способ

хранения данных, сохранив методы их обработки, и при этом остальные объекты смогут работать с соответствующими объектами так же, как и прежде.

Еще один принцип ООП облегчает разработку собственных классов в языке Java: класс можно сконструировать на основе других классов. В этом случае говорят, что вновь созданный класс *расширяет* класс, на основе которого он создан. Язык Java, по существу, создан на основе "глобального суперкласса", называемого `Object`. Все остальные объекты расширяют его. В следующей главе мы рассмотрим этот вопрос подробнее.

Если вы разрабатываете класс на основе существующего, то новый класс содержит все свойства и методы расширяемого класса, кроме того, в нем добавляются новые методы и поля данных. Расширение класса и получение на его основе нового называется *наследованием* (inheritance). Детально принцип наследования будет описан в следующей главе.

Объекты

В объектно-ориентированном программировании определены следующие ключевые свойства объектов.

- *Поведение* (behavior) объекта – что с ним можно делать и какие методы к нему можно применять.
- *Состояние* объекта – как этот объект реагирует на применение методов.
- *Сущность* (identity) объекта – чем данный объект отличается от других, характеризующихся таким же поведением и состоянием.

Все объекты, являющиеся экземплярами одного и того же класса, *ведут* себя одинаково. Поведение объекта определяется методами, которые можно вызвать.

Далее, каждый объект сохраняет информацию о своем *состоянии*. Со временем состояние объекта может измениться, однако спонтанно это произойти не может. Состояние объекта может изменяться только в результате вызовов методов. (Если состояние объекта изменилось вследствие иных причин, значит, принцип инкапсуляции не соблюден.)

Состояние объекта не полностью описывает его, поскольку каждый объект имеет свою собственную *сущность*. Например, в системе обработки заказов два заказа могут отличаться друг от друга, даже если они относятся к одним и тем же товарам. Заметим, что индивидуальные объекты, представляющие собой экземпляры класса, *всегда* отличаются своей сущностью и *обычно* отличаются своим состоянием.

Эти основные характеристики могут влиять друг на друга. Например, состояние объекта может влиять на его поведение. (Если заказ "выполнен" или "оплачен", объект может отказаться выполнить вызов метода, требующего добавить или удалить изделие. И наоборот, если заказ "пуст", т.е. ни одна единица товара не была заказана, он не может быть выполнен.)

В традиционной процедурной программе выполнение начинается сверху, с функции `main`. При разработке объектно-ориентированной системы понятие "верха" не существует, и новички в ООП часто интересуются, с чего начинать. Ответ таков: "Сначала найдите классы и добавьте методы в каждый класс".

Классы представляют собой аналоги имен существительных в описании решаемой задачи, а методы — используемых при этом глаголов.

Например, при описании системы обработки заказов используются следующие имена существительные:

- изделие;
- заказ;
- адрес доставки;
- оплата;
- счет.

Этим именам соответствуют классы `Item`, `Order` и т.д.

Рассмотрим теперь глаголы. Предметы *заказываются*. Заказы *выполняются* или *отменяются*. Оплата заказа *осуществляется*. Используя эти глаголы, можно определить объект, выполняющий такие действия. Например, если поступил новый заказ, то ответственность за его обработку должен нести объект `Order`, поскольку именно в нем содержится информация о способе хранения и видах заказываемых предметов. Следовательно, в классе `Order` должен существовать метод `add`, получающий объект `Item` в качестве параметра.

Разумеется, "правило имен существительных и глаголов" – это не более чем совет, и только опыт может помочь программисту решить, какие существительные и глаголы важны при создании класса.

Отношения между классами

Между классами существуют три обычных отношения.

- *Зависимость* ("использует").
- *Агрегирование* ("содержит").
- *Наследование* ("является").

Отношение зависимости (dependence – "uses-a") наиболее очевидное и распространенное. Предположим, например, что в системе поддержки заказов используются классы `Order` и `Account`. Класс `Order`, вероятнее всего, использует класс `Account`, поскольку объекты `Order` должны иметь доступ к объектам `Account`, чтобы проверить кредитоспособность заказчика. Однако класс `Item` не зависит от класса `Account`, так как объекты `Item` никогда не интересуются состоянием счета заказчика. Следовательно, класс зависит от другого класса, если его методы выполняют какие-либо действия с экземплярами этого класса.

Старайтесь минимизировать количество взаимозависимых классов. Если класс `A` не знает о существовании класса `B`, он тем более ничего не знает о любых его изменениях! (Это значит, что любые изменения класса `B` не повлияют на поведение обектов класса `A`.)

Отношение агрегирования (aggregation – "has-a") понять довольно просто. Оно означает, что класс `A` содержит экземпляры класса `B`. Например, объект `Order` может содержать экземпляры класса `Item`.

Наследование (inheritance — "is-a") выражает отношение между более конкретным и более общим классом. Например, класс `RushOrder` является производным от класса `Order`. Класс `RushOrder` имеет специальные методы для обработки приоритетов и разные методы для вычисления стоимости доставки, в то время как другие его методы, например для заказа предмета и выписывания счета, унаследованы от класса `Order`. В частности, если класс `A` расширяет класс `B`, то говорят, что класс `A` наследует методы класса `B` и, кроме них, имеет дополнительные возможности. (Более подробно наследование описано в следующей главе.)

Многие программисты используют средства UML (Unified Modeling Language) для изображения *диаграмм классов* (class diagrams), описывающих отношения между классами. Пример такой диаграммы приведен на рис. 4.1. Здесь классы изображены с помощью прямоугольников, а отношения между ними — различными стрелками. В табл. 4.1 показаны основные обозначения, принятые в языке UML.

Таблица 4.1. Обозначение отношений между классами, принятые в языке UML

Отношение	Обозначение в языке UML
Наследование	
Зависимость	
Агрегирование	
Связь	
Направленная связь	

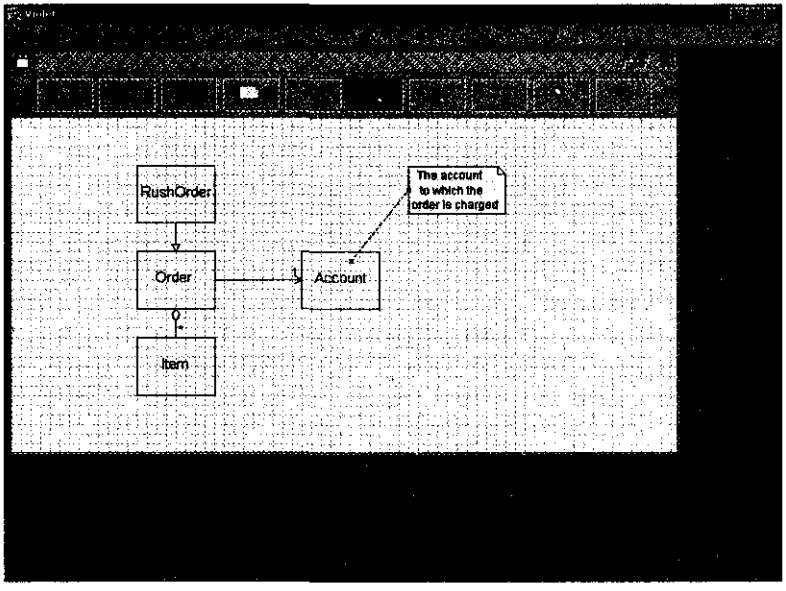


Рис. 4.1. Диаграмма классов

Некоторые специалисты не признают понятие агрегирования и предпочитают использовать отношение "связи" ("association"). С точки зрения моделирования это разумно. Однако для программистов отношение, при котором один объект содержит другой, гораздо удобнее. Мы предпочитаем использовать понятие агрегирования еще по одной причине: его обозначение проще для восприятия, чем обозначение отношения связи (табл. 4.1).

Для создания диаграмм на языке UML разработано много инструментов. Некоторые поставщики предлагают очень мощные (и очень дорогие) программы, призванные стать основным средством проектирования. Среди них можно отметить продукты Rational Rose (`http://www.ibm.com/software/awdtools/developer/modeler`) и Together (`http://www.borland.com/together`). В качестве альтернативы можно назвать программу ArgoUML, созданную на принципах открытого кода (`http://argouml.tigris.org/`). Коммерческая версия программы поддержки URL доступна по адресу `http://gentleware.com`. Если вам нужно лишь нарисовать диаграмму, приложив минимум усилий, попробуйте воспользоваться программой Violet (`http://horstmann.com/violet`).

Различия между ООП и традиционным процедурным программированием

Завершим наше краткое введение в объектно-ориентированное программирование анализом различий между ним и процедурной моделью. В процедурном программировании сначала формулируется задача, а затем выполняются следующие действия.

- Путем пошагового уточнения исходная задача разбивается на все более мелкие подзадачи, пока они не станут настолько простыми, что их можно будет реализовать непосредственно (программирование "сверху вниз").
- Создаются процедуры, предназначенные для решения простых задач; затем эти процедуры объединяются в более сложные, пока программа не станет делать именно то, что требуется (программирование "снизу вверх").

Большинство программистов, конечно, используют смешанные стратегии "сверху вниз" и "снизу вверх". При разработке процедур нужно придерживаться такого же правила, как и при создании методов в объектно-ориентированном программировании: ищите глаголы, или действия, в описании задачи. Существенное отличие заключается в том, что при объектно-ориентированном программировании в проекте *сначала* выделяются классы и лишь затем определяются их методы. Между традиционными процедурами и ООП есть еще одна существенная разница: каждый метод связан с классом и класс отвечает за его выполнение.

Для решения небольших задач процедурный подход вполне оправдан. Однако при разработке больших проектов классы и методы имеют существенные преимущества. Классы предоставляют удобный механизм кластеризации методов. Даже несложный Web-браузер для своей реализации может потребовать около 2000 функций, либо 100 классов, содержащих в среднем по 20 методов. Программистам намного легче оперировать с набором классов. Кроме того, в этом случае проще распределить задачи между участниками группы разработчиков. Немаловажную роль в организации совместной работы играет инкапсуляция: классы скрывают детали представления данных от всех остальных классов. Как показано на рис. 4.2, если при разработке допущена ошибка, которая приводит к искажению данных, причину легче найти среди 20 классов, чем среди 2000 процедур.

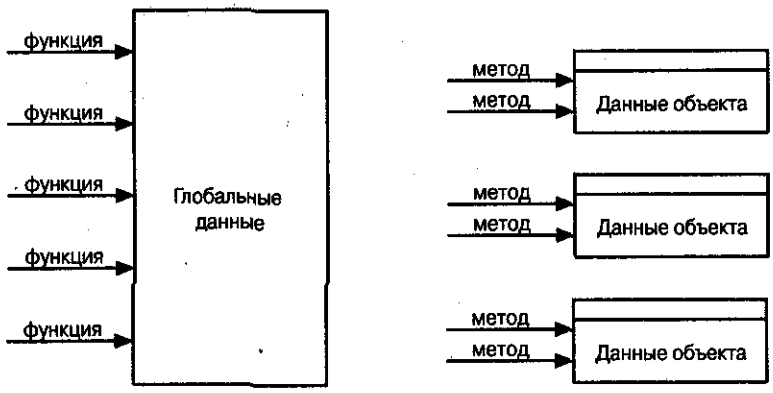


Рис. 4.2. Процедурный и объектный подход

Можно возразить, что принцип формирования объектов не слишком отличается от *принципа модульности* (modularization). Наверняка вам приходилось писать программы, разбивая их на модули, обменивающиеся информацией исключительно с помощью вызова процедур, не прибегая к совместному использованию данных (sharing data). *Если правильно пользоваться этим приемом, можно обеспечить инкапсуляцию* данных. Однако малейшее отклонение от описанного подхода открывает доступ к данным из другого модуля — в этом случае об инкапсуляции говорить не приходится.

Существует более серьезная проблема: в то время как на основе класса можно создать несколько объектов с одинаковым поведением, в процедурно-ориентированных языках невозможно получить несколько копий одного модуля. Допустим, мы достигли модульной инкапсуляции, объединив совокупность заказов вместе с прекрасным сбалансированным бинарным деревом, обеспечивающим быстрый доступ. И вдруг выясняется, что нужно иметь *две* такие совокупности: одну для текущих заказов, а другую — для выполненных. Нельзя просто дважды вызвать модуль, реализующий дерево заказов. Для этого нужно скопировать и переименовать все процедуры! Классы не имеют таких ограничений. Единожды определив класс, легко создать любое количество его экземпляров (в то время как модуль существует только в одном экземпляре).

В данном разделе мы лишь слегка затронули очень большую тему. В конце этой главы помещен короткий раздел "Советы по разработке классов", однако для более глубокого понимания процесса объектно-ориентированного проектирования нужно изучать книги, посвященные ООП и UML. В частности, мы можем порекомендовать книгу Гради Буча (Grady Booch), Айвара Джекобсона (Ivar Jacobson) и Джеймса Рамбо (James Rumbaugh) *Unified Modelling Landuage User Guide* (Addison-Wesley, 1999).

Использование готовых классов

Поскольку на языке Java ничего нельзя сделать без классов, мы вкратце обсудили в предыдущих разделах, как работают некоторые из них. К сожалению, в Java есть классы, к которым не совсем подходят приведенные выше рассуждения. Ярким примером является класс `Math`. Как вы уже видели, методы класса `Math`, например метод `Math.random`, можно вызывать, ничего не зная о деталях их реализации. Для обращения к методу достаточно знать его имя и параметры (если они предусмотрены). К сожалению, в классе `Math` нет данных, а следовательно, нет необходимости в их со-

крытии. Таким образом, программист может не заботиться о создании объектов и инициализации их полей, – в классе Math ничего подобного нет!

В следующем разделе мы рассмотрим класс Date. Мы увидим, как создаются экземпляры этого класса и вызываются методы.

Объекты и объектные переменные

Чтобы работать с объектами, их нужно сначала создать и задать их исходное состояние. Затем к этим объектам можно применять методы.

В языке Java для создания новых экземпляров используются *конструкторы*. Конструктор – это специальный метод, предназначенный для создания и инициализации экземпляра класса. В качестве примера можно привести класс Date, содержащийся в стандартной библиотеке Java. С помощью объектов этого класса можно описать текущий или любой другой момент времени, например "December 31, 1999, 23:59:59 GMP".

Может возникнуть вопрос: почему для представления даты и времени применяются классы, а не встроенные типы (как в некоторых языках)? Такой подход применяется, например, в Visual Basic; при этом дата задается в виде #6/1/1995#. На первый взгляд, это удобно — программист может использовать встроенный тип и не заботиться о классах. Однако не является ли удобство кажущимся? В некоторых странах даты записываются в виде месяц/день/год, а в других — год/месяц/число. Могут ли разработчики языка предусмотреть все возможные варианты? Даже если это удастся сделать, соответствующие средства будут слишком сложны, причем программисты будут вынуждены применять их. Использование классов позволяет переложить ответственность за решение этих проблем с разработчиков языка на создателей библиотек. Если системный класс плох, то разработчики всегда могут написать свой собственный.

Имя конструктора всегда совпадает с именем класса. Следовательно, конструктор класса Date называется Date. Чтобы создать объект Date, конструктор нужно объединить с оператором new, как показано в следующем примере:

```
new Date()
```

Это выражение создает новый объект, который инициализируется текущими датой и временем.

При желании объект можно передать методу.

```
System.out.println(new Date());
```

И наоборот, можно вызвать метод вновь созданного объекта. Среди методов класса Date есть метод toString(), позволяющий представить дату в виде строки. Его можно вызвать следующим образом:

```
String = new Date().toString();
```

В этих двух примерах созданный объект использовался только один раз. Обычно объект приходится использовать в дальнейшем. Чтобы это стало возможным, необходимо связать объект с некоторым идентификатором, другими словами, присвоить объект переменной.

```
Date birthday = new Date();
```

На рис. 4.3 условно показана переменная birthday, *ссылающаяся на вновь созданный объект.*

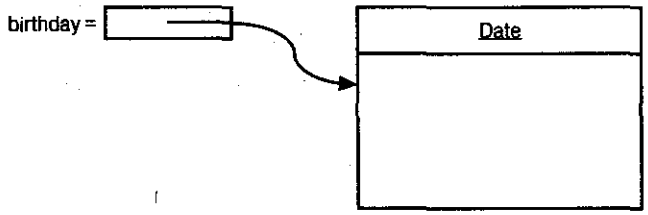


Рис. 4.3. Создание нового объекта

Между объектами и объектными переменными есть существенная разница. Например, приведенное ниже выражение определяет объектную переменную `deadline`, которая может ссылаться на объекты типа `Date`.

```
Date deadline; // Переменная deadline не ссылается ни на один объект
```

Важно понимать, что на данном этапе сама переменная `deadline` *объектом не является* и даже не ссылается ни на один объект. Поэтому ни один метод класса `Date` с помощью этой переменной вызывать пока нельзя. Попытка сделать это приведет к появлению сообщения об ошибке.

```
s = deadline.toString(); // Вызывать метод еще рано
```

Сначала переменную `deadline` нужно инициализировать. У программиста есть две возможности. Разумеется, переменную можно инициализировать вновь созданным объектом:

```
deadline = new Date();
```

Кроме того, можно заставить переменную ссылаться на существующий объект:

```
deadline = birthday;
```

Теперь переменные `deadline` и `birthday` ссылаются на *один и тот же* объект (рис. 4.4).

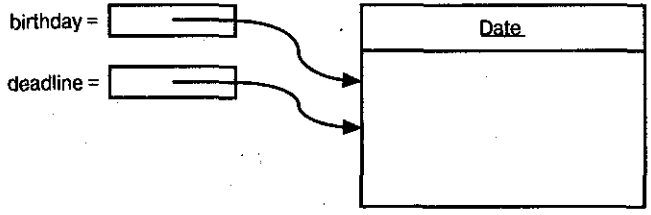


Рис. 4.4. Объектные переменные, ссылающиеся на один и тот же объект

Важно помнить, что объектная переменная фактически не содержит никакого объекта. Она лишь *ссылается* на него.

В языке Java значение любой объектной переменной представляет собой ссылку на объект, размещенный в другом месте. Оператор `new` также возвращает ссылку. Например, приведенная ниже строка кода состоит из двух частей.

```
Date deadline = new Date();
```

Выражение `new Date()` создает объект типа `Date`, а значение переменной представляет собой ссылку на вновь созданный объект.

Объектной переменной можно явно присвоить ссылку `null`, чтобы отметить тот факт, что она пока не ссылается ни на один объект.

```
deadline = null;
...
if(deadline != null)
   System.out.println(deadline);
```

Если вы попытаетесь вызывать метод объекта посредством переменной, значение которой равно `null`, то при выполнении программы возникнет ошибка.

```
birthday = null;
String s = birthday.toString(); // Ошибка!
```

Локальные объектные переменные не инициализируются автоматически. Программист должен сам инициализировать переменную, либо вызвав оператор `new`, либо присвоив значение `null`.

Многие ошибочно полагают, что объектные переменные в языке Java похожи на *переменные языка C++, типы которых задаются именами классов. Однако в языке C++* такая переменная не может иметь значение `null`. Кроме того, им ничего нельзя присваивать. Объектные переменные в языке Java следует считать аналогами *указателей на объекты*. Например, выражение

```
Date birthday; // Java
```

почти эквивалентно приведенной ниже строке кода.

```
Date* birthday; // C++
```

Такая аналогия все расставляет по своим местам. Разумеется, указатель `Date*` не *инициализируется, пока не выполнен оператор* `new`. *Синтаксис выражений в языках* C++ и Java почти совпадает.

```
Date* birthday = new Date(); // C++
```

При копировании одной переменной в другую в обеих переменных появляется ссылка на *один и тот же объект. Эквивалентом ссылки* `null` *в языке C++ является указатель* `NULL`.

Все объекты в языке Java располагаются в динамической памяти. Если объект содержит другую объектную переменную, она представляет собой всего лишь указатель на другой объект, расположенный в области памяти под названием "куча".

В языке C++ указатели доставляют много хлопот, так как из-за них часто возникают ошибки. Очень легко создать неверный указатель или потерять управление памятью. В языке Java эти проблемы попросту не существуют. Если вы используете неинициализированный указатель, то система поддержки выполнения программ обязательно сообщит об ошибке и не продолжит выполнение некорректной программы, выдавая случайные результаты. Программист может не заботиться об управлении памятью, поскольку механизм "сборки мусора" делает это за него автоматически.

В языке C++ большое внимание уделено автоматическому копированию объектов. Например, копией связного списка является новый связный список, который, имея старое содержимое, содержит совершенно другие связи. Другими словами, копирование объектов осуществляется так же, как и копирование встроенных типов. В языке Java для получения полной копии объекта используется метод `clone`.

Класс GregorianCalendar из библиотеки Java

В предыдущих примерах мы использовали класс `Date`, являющийся частью стандартной библиотеки Java. Экземпляр класса `Date` находится в состоянии, которое отражает *конкретный момент времени*.

Хотя при использовании класса `Date` нам необязательно знать о формате даты, отметим, что время представляется количеством миллисекунд (положительным или отрицательным), отсчитанным от фиксированного момента времени, так называемого *начала эпохи*, т.е. от 00:00:00 UTC, 1 января 1970 года. Аббревиатура UTC означает Universal Coordinated Time – научный стандарт времени. UTC применяется наряду с более известным GMT (Greenwich Mean Time).

Однако класс `Date` не очень удобен для работы с датами. Разработчики библиотеки Java считали, что представление даты, например "December 31, 1999, 23:59:59", является совершенно произвольным и должно зависеть от *календаря*. Данное конкретное представление подчиняется григорианскому календарю, самому распространенному календарю в мире. Однако тот же самый момент времени совершенно иначе представляется в китайском или еврейском лунном календаре, не говоря уже о календаре, которым будут пользоваться потенциальные заказчики с Марса.

Вся история развития человечества сопровождалась созданием календарей — систем именования различных моментов времени. Как правило, основой для календарей был солнечный или лунный цикл. Если вас интересуют подобные вопросы, обратитесь к книге Наума Дершовица (Nachum Dershowitz) и Эдварда М. Рейнголда (Edward M. Reingold) *Calendrical Calculations* (Cambridge University Press, 2nd ed., 2001). Там вы найдете сведения о календаре французской революции, календаре Майя и других экзотических системах.

Разработчики библиотеки решили отделить вопросы, связанные с отслеживанием моментов времени, от вопросов их представления. Таким образом, стандартная библиотека Java содержит два отдельных класса: класс `Date`, представляющий момент времени, и класс `GregorianCalendar`, расширяющий более общий класс `Calendar`, описывающий свойства календаря в целом. Теоретически можно расширить класс `Calendar` и реализовать китайский лунный или марсианский календари. Однако в стандартной библиотеке, кроме григорианского календаря, пока нет никакой другой реализации.

Отделение измерения времени от календарей представляет собой хорошее решение, вполне соответствующее принципу объектно-ориентированного программирования.

Класс `Date` содержит лишь небольшое количество методов, позволяющих сравнивать два момента времени. Например, методы `before` и `after` определяют, предшествует один момент времени другому или следует за ним.

```
if (today.before(birthday))
   System.out.println("Еще есть время купить подарок.");
```

Класс `GregorianCalendar` содержит гораздо больше методов, чем класс `Date`. В частности, в нем есть несколько полезных конструкторов. Так, приведенное ниже выражение формирует новый объект, представляющий дату и момент времени, когда он был создан.

```
new GregorianCalendar()
```

На самом деле класс `Date` имеет такие методы, как `getDay()`, `getMonth()` и `getYear()`, но использовать их без крайней необходимости *не рекомендуется*.

Эти методы были частью класса `Date` еще до того, как разработчики библиотеки поняли, что классы, реализующие разные календари, разумнее было бы отделить друг от друга. Сделав это, они пометили методы класса `Date` как *не рекомендованные к применению* (deprecated). Вы можете продолжать использовать их в своих программах, получая при этом предупреждения от компилятора, а еще лучше их не применять совсем, поскольку в будущем они могут быть удалены из библиотеки.

Вы можете создать объект, представляющий полночь даты, указанной как год, месяц и день.

```
new GregorianCalendar(1999, 11, 31)
```

Довольно странно, что количество месяцев отсчитывается от нуля. Так, число 11 означает декабрь. Для большей ясности можно использовать константу `Calendar.December`.

```
new GregorianCalendar(1999, Calendar.December, 31)
```

Кроме того, посредством конструктора `GregorianCalendar` можно задать время.

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31, 23, 59, 59)
```

Чаще всего ссылка на созданный объект присваивается переменной.

```
GregorianCalendar deadline = new GregorianCalendar(...);
```

В классе `GregorianCalendar` инкапсулированы поля экземпляра, в которых записана указанная дата. Не имея доступа к исходному тексту, невозможно определить, какое представление даты и времени использует этот класс. Разумеется, для программиста, применяющего готовый класс, это совершенно неважно. Важно лишь то, какие методы доступны ему.

Модифицирующие методы и методы доступа

После того как вы познакомились с классом `GregorianCalendar`, у вас, возможно, возникли вопросы: как получить текущий день, месяц или год на базе даты, инкапсулированной в объекте данного класса? Каким образом можно изменить эти значения? Ответы на эти вопросы вы найдете в данном разделе, а дополнительную информацию — в документации по API. Рассмотрим наиболее важные методы.

Календарь должен вычислять атрибуты, соответствующие указанному моменту времени, например, дату, день недели, месяц или год. Получить одно из этих значений можно, используя метод `get()` класса `GregorianCalendar`. Чтобы выбрать желаемый атрибут, нужно передать методу константу, определенную в классе `Calendar`, например `Calendar.MONTH` или `Calendar.DAY_OF_WEEK`:

```
GregorianCalendar now = new GregorianCalendar();
int month = now.get(Calendar.MONTH);
int weekday = now.get(Calendar.DAY_OF_WEEK);
```

В конце данного раздела перечислены все константы, которые можно использовать для получения требуемой информации из объекта `GregorianCalendar`.

Состояние объекта можно изменить с помощью метода `set()`.

```
deadline.set(Calendar.YEAR, 2001);
deadline(Calendar.MONTH, Calendar.APRIL);
deadline.set(Calendar.DAY, 15);
```

Существует способ установить год, месяц и день с помощью одного вызова.

```
deadline.set(2001, Calendar.APRIL, 15);
```

При желании вы можете добавить к заданной дате определенное количество дней, недель, месяцев и т.д.

```
deadline.add(Calendar.MONTH, 3); // Сдвинуть момент deadline на 3 месяца.
```

Добавив отрицательное число, вы сдвинете момент времени, описываемый объектом, назад.

Между методом `get()`, с одной стороны, и методами `set()` и `add()`, с другой, есть принципиальная разница. Метод `get()` только просматривает состояние объекта и сообщает о нем, в то время как методы `set()` и `add()` модифицируют состояние объекта. Методы, которые могут изменять поля экземпляра, называются *модифицирующими* (mutator), а методы, которые могут лишь просматривать поля экземпляра, не изменяя их, называются *методами доступа* (accessor).

В языке C++ для формирования метода доступа используется суффикс `const`. Метод, не объявленный с помощью ключевого слова `const`, считается модифицирующим. Однако в языке Java нет специальных синтаксических средств, позволяющих различать модифицирующие методы и методы доступа.

В именах методов доступа принято использовать префикс `get()`, а в именах модифицирующих методов – префикс `set()`. Например, класс `GregorianCalendar` обеспечивает получение и установку момента времени, представленного объектом, посредством методов `getTime()` и `setTime()`.

```
Date time = calendar.getTime();
calendar.setTime(time);
```

Эти методы очень полезны для преобразований объектов `GregorianCalendar` в объекты `Date` и наоборот. Предположим, что нам известны год, месяц и день и мы хотим создать объект `Date`, поля которого были бы заполнены этими значениями. Поскольку класс `Date` ничего не знает о календарях, создадим сначала объект `GregorianCalendar`, а затем вызовем метод `getTime()` для получения даты.

```
GregorianCalendar calendar
   = new GregorianCalendar(year, month, day);
Date hireDay = calendar.getTime();
```

И наоборот, если нужно определить год, месяц или день, записанные в объекте `Date`, создадим объект `GregorianCalendar`, установим время, а затем вызовем метод `get()`.

```
GregorianCalendar calendar = new GregorianCalendar();
calendar.setTime(hireDay);
int year = calendar.get(Calendar.YEAR);
```

В конце этого раздела приведена программа, иллюстрирующая работу класса GregorianCalendar. Программа выводит на экран календарь текущего месяца в следующем формате:

```
Sun  Mon  Tue  Wed  Thu  Fri  Sat
                              1
 2    3    4    5    6    7    8
 9   10   11   12   13   14   15
16   17   18   19*  20   21   22
23   24   25   26   27   28   29
30   31
```

Текущий день помечен звездочкой, причем программа знает, как вычислять дни недели.

Рассмотрим ключевые вопросы, связанные с работой этой программы. Сначала создадим объект GregorianCalendar, инициализированный текущими датой и временем. (На самом деле в данном приложении время нас не интересует.)

```
GregorianCalendar d = new GregorianCalendar();
```

Затем определим текущий день и месяц, дважды вызвав метод get().

```
int today = d.get(Calendar.DAY_OF_MONTH);
int month = d.get(Calendar.MONTH);
```

После этого передадим методу set() объекта d параметр, задающий первое число месяца, и получим день недели, соответствующий этой дате.

```
d.set(Calendar.DAY_OF_MONTH, 1);
int weekday = d.get(Calendar.DAY_OF_WEEK);
```

Переменная weekday будет содержать значение 1 (или Calendar.SUNDAY), если первый день месяца — воскресенье, 2 (или Calendar.MONDAY) — если понедельник, и т.д.

Затем на экран выводится заголовок и пробелы, выравнивающие первую строку календаря.

Для каждого дня выведем на экран пробел, если число меньше 10, затем — само число, а потом звездочку, если число соответствует сегодняшнему дню. Каждое воскресенье выводится с новой строки.

Затем установим объект d на следующий день.

```
d.add(Calendar.DAY_OF_MONTH, 1);
```

Когда нам следует остановиться? Нам неизвестно, сколько в месяце дней: 31, 30, 29 или 28. Мы продолжаем итерации, пока объект d остается в пределах текущего месяца.

```
do
{
   ...
}
while (d.get(Calendar.MONTH) == month);
```

Как только объект d перейдет на следующий месяц, программа завершит свою работу.

Полный текст программы приведен в листинге 4.1.

Как видим, класс GregorianCalendar позволяет легко создавать программы для работы с календарем, выполняя такие сложные действия, как отслеживание дней недели и учет продолжительности месяцев. Программист не обязан ничего знать о том,

как именно класс `GregorianCalendar` вычисляет месяцы и дни недели. Нужно просто использовать *интерфейс* класса — методы `get()`, `set()` и `add()`.

Основное назначение этой программы — показать, как можно использовать интерфейс класса для выполнения действительно сложных задач, не вникая в детали реализации.

Листинг 4.1. Содержимое файла CalendarTest.java

```
import java.util.*;

public class CalendarTest
{
   public static void main(String[] args)
   {
      // Создание объекта d, представляющего текущую дату.
      GregorianCalendar d = new GregorianCalendar();

      int today = d.get(Calendar.DAY_OF_MONTH);
      int month = d.get(Calendar.MONTH);

      // Установка объекта d на первое число месяца.
      d.set(Calendar.DAY_OF_MONTH, 1);

      int weekday = d.get(Calendar.DAY_OF_WEEK);

      // Вывод заголовка таблицы.
      System.out.println("Sun  Mon  Tue  Wed  Thu  Fri  Sat");

      // При отображении первой строки календаря
      // возможен сдвиг.
      for (int i = Calendar.SUNDAY; i < weekday; i++ )
         System.out.print("     ");

      do
      {
         // Вывод дня месяца.
         int day = d.get(Calendar.DAY_OF_MONTH);
         System.out.printf("%3d", day);

         // Символом * помечается текущий день.
         if (day == today)
            System.out.print("*");
         else
            System.out.print(" ");

         // После каждой субботы начинается новая строка.
         if (weekday == Calendar.SATURDAY)
            System.out.println();

         // Перевод d на следующий день.
         d.add(Calendar.DAY_OF_MONTH, 1);
         weekday = d.get(Calendar.DAY_OF_WEEK);
      }
      while (d.get(Calendar.MONTH) == month);
      // Цикл завершается, когда объект d установлен
      // на первый день следующего месяца.
```

```
        // При необходимости переводится строка.
        if (weekday != Calendar.SUNDAY)
            System.out.println();
    }
}
```

Чтобы не усложнять программу, имена выводятся на английском языке, а первым днем недели считается воскресенье. Если вы хотите выводить календарь по соглашениям другой страны, ознакомьтесь с классом `DateFormatSymbols`. Метод `Calendar.getFirstDayOfWeek()` возвращает первый день недели. Для США — это воскресенье, а для Германии — понедельник.

- `GregorianCalendar()`

  Создает объект `GregorianCalendar`, представляющий текущее время для часового пояса и страны, заданных по умолчанию.

- `GregorianCalendar(int year, int month, int date)`

  Создает объект `GregorianCalendar`, представляющий заданную дату.

  Параметры

`year`	Год
`month`	Месяц. Отсчет начинается с нуля. Например, 0 — это январь
`date`	Число месяца.

- `GregorianCalendar(int year, int month, int date, int hour, int minutes, int seconds)`

  Создает объект `GregorianCalendar`, представляющий указанное время для часового пояса и страны, заданных по умолчанию.

  Параметры

`year`	Год
`month`	Месяц. Отсчет начинается с нуля. Например, 0 — это январь
`date`	Число месяца
`hour`	Час (от 0 до 23)
`minutes`	Минуты (от 0 до 59)
`seconds`	Секунды (от 0 до 59)

- `int get(int field)`

  Возвращает значение заданного поля.

  Параметр `field` — Одна из констант:
  `Calendar.ERA`, `Calendar.YEAR`,
  `Calendar.WEEK_OF_YEAR`,
  `Calendar.WEEK_OF_MONTH`,
  `Calendar.DAY_OF_MONTH`,
  `Calendar.DAY_OF_YEAR`, `Calendar.DAY_OF_WEEK`,
  `Calendar.DAY_OF_WEEK_IN_MONTH`,

`Calendar.AM_PM, Calendar.HOUR, Calendar.HOUR_OF_DAY, Calendar.MINUTE, Calendar.SECOND, Calendar.MILLISECOND, Calendar.ZONE_OFFSET` или `Calendar.DST_OFFSET`

- `void set(int field, int value)`

  Устанавливает значения заданных полей.

  Параметры `field` — Одна из констант, допустимых для метода `get`
  `value` — Новое значение

- `void set(int year, int month, int day)`

  Устанавливает новую дату в объекте.

  Параметры `year` — Год
  `month` — Месяц. Отсчет начинается с нуля. Например, 0 — это январь
  `day` — Число месяца

- `void set(int year, int month, int day, int hour, int minutes, int seconds)`

  Устанавливает дату и время в новом объекте.

  Параметры `year` — Год
  `month` — Месяц. Отсчет начинается с нуля. Например, 0 — это январь
  `day` — Число месяца
  `hour` — Час (от 0 до 23)
  `minutes` — Минуты (от 0 до 59)
  `seconds` — Секунды (от 0 до 59)

- `void add(int field, int amount)`

  Арифметический метод для работы с датой и временем. Добавляет к полям заданную величину. Например, чтобы прибавить 7 дней к текущей дате, нужно сделать вызов `c.add(Calendar.DAY_OF_MONTH, 7)`.

  Парметры `field` — Изменяемое поле (должно содержать значение одной из констант, предусмотренных для метода `get`)
  `amount` — Добавляемое время (может быть отрицательным числом)

- `void setTime(Date time)`

  Устанавливает заданный момент времени.

  Параметр `time` — Момент времени

- `Date getTime()`

  Возвращает момент времени, представленный текущим значением объекта.

Определение собственных классов

В главе 3 мы уже попробовали создавать простые классы. Однако все они состояли из одного-единственного метода `main()`. Теперь пришло время научиться создавать "рабочие" классы для более сложных приложений. Обычно в этих классах нет метода `main()`. Вместо этого у них есть другие методы и поля. Чтобы написать полностью законченную программу, нужно объединить классы, один из которых содержит метод `main()`.

Класс Employee

Простейшее определение класса в языке Java имеет следующий вид:

```
class ИмяКласса
{
   конструктор_1
   конструктор_2
   ...
   метод_1
   метод_2
   ...
   поле_1
   поле_2
   ...
}
```

Мы придерживаемся стиля, согласно которому сначала указываются методы класса, а в конце — его поля. Это сделано для того, чтобы привлечь основное внимание к интерфейсу, а не к реализации класса.

Рассмотрим следующую, весьма упрощенную версию класса `Employee`, который можно использовать для создания платежной ведомости.

```
class Employee
{
   // Конструктор
   public Employee(String n, double s,
      int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar
         = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   // Метод 1
   public String getName()
   {
      return name;
   }

   // Другие методы
   ...
```

```
   // Поля экземпляра

   private String name;
   private double salary;
   private Date hireDay;
}
```

Детали реализации этого класса мы проанализируем в следующих разделах. А сейчас рассмотрим код, представленный в листинге 4.2 и иллюстрирующий работу класса Employee.

В данной программе создается массив Employee, в который заносятся три объекта Employee.

```
Employee[] staff = new Employee[3];

staff[0] = new Employee("Carl Cracker", . . .);
staff[1] = new Employee("Harry Hacker", . . .);
staff[2] = new Employee("Tony Tester", . . .);
```

Затем, для того чтобы поднять зарплату каждого сотрудника на 5%, вызывается метод класса Employee.

```
for (Employee e : staff)
   e.raiseSalary(5);
```

В заключение с помощью методов getName(), getSalary() и getHireDay() выводится информация о каждом сотруднике.

```
for (Employee e : staff)
{
   System.out.println("name=" + getName()
      + ", зарплата=" + e.getSalary()
      + ", дата найма на работу=" + e.getHireDay());
}
```

Заметим, что этот пример программы состоит из *двух* классов: Employee и EmployeeTest, объявленного как public. Метод main() с выражениями, которые мы только что описали, содержится в классе EmployeeTest.

Исходный текст программы содержится в файле EmployeeTest.java, *поскольку его* имя должно совпадать с именем общедоступного класса. В исходном файле может быть только один класс, объявленный как public, и любое количество классов, в объявлении которых данное ключевое слово отсутствует.

Далее при компиляции исходного кода компилятор создает два файла классов: EmployeeTest.class и Employee.class.

Выполнение программы начинается, когда интерпретатор получает имя класса, содержащего метод main().

```
java EmployeeTest
```

При этом интерпретатор начинает обрабатывать метод main() из класса EmployeeTest. В результате выполнения кода создаются три новых объекта Employee и отображается их состояние.

Листинг 4.2. Содержимое файла EmployeeTest.java

```
import java.util.*;

public class EmployeeTest
{
   public static void main(String[] args)
   {
      // Заполнение массива staff тремя объектами.
      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Carl Cracker", 75000,
          1987, 12, 15);
      staff[1] = new Employee("Harry Hacker", 50000,
          1989, 10, 1);
      staff[2] = new Employee("Tony Tester", 40000,
          1990, 3, 15);

      // Увеличение зарплаты на 5%
      for (Employee e : staff)
         e.raiseSalary(5);

      // Вывод информации о всех объектах Employee
      for (Employee e : staff)
         System.out.println("name=" + e.getName()
            + ",salary=" + e.getSalary()
            + ",hireDay=" + e.getHireDay());
   }
}

class Employee
{
   public Employee(String n, double s,
       int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new
     GregorianCalendar(year, month - 1, day);
         // В классе GregorianCalendar январю соответствует 0.
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }
```

```
   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   private String name;
   private double salary;
   private Date hireDay;
}
```

Использование нескольких исходных файлов

Программа, приведенная в листинге 4.2, содержит два класса в одном исходном файле. Многие программисты предпочитают помещать каждый класс в отдельный файл. Например, класс `Employee` можно разместить в файле `Employee.java`, а класс `EmployeeTest` — в файле `EmployeeTest.java`.

Существуют различные способы компиляции программы, код которой содержится в двух исходных файлах. Вы можете использовать символ групповой операции, например:

```
javac Employee*.java
```

В результате все исходные файлы, имена которых соответствуют указанному вами шаблону, будут скомпилированы в файлы классов. Можно также ограничиться приведенной ниже командой.

```
javac EmployeeTest.java
```

Как ни странно, файл `Employee.java` также будет скомпилирован. Обнаружив, что в файле `EmployeeTest.java` используется класс `Employee`, компилятор языка Java станет искать файл `Employee.class`. Если он его не найдет, то автоматически будет скомпилирован файл `Employee.java`. Более того, если файл `Employee.java` создан позже, чем существующий файл `Employee.class`, компилятор языка Java также выполнит повторную компиляцию и создаст файл класса.

Если вы знакомы с утилитой `make`, имеющейся в Unix и других операционных системах, то поведение компилятора не вызовет вопросов. Дело в том, что в компиляторе языка Java реализованы возможности этой утилиты.

Анализ класса Employee

Проанализируем класс `Employee`. Начнем с методов этого класса. Изучая исходный текст программы, легко видеть, что в классе `Employee` реализованы один конструктор и четыре метода.

```
public Employee(String n, double s,int year, int month, int day)
public String getName()
public double getSalary()
public Date getHireDay()
public void raiseSalary(double byPercent)
```

Все методы этого класса объявлены как `public`, т.е. обращение к этим методам *может осуществляться из любого класса*. (Существуют четыре возможных уровня доступа; они рассматриваются в этой и следующей главах.)

Отметим также, что в классе есть три поля экземпляра для хранения данных, обрабатываемых внутри объекта Employee.

```
private String name;
private double salary;
private Date hireDay;
```

Ключевое слово private означает, что к данным полям имеют доступ только методы самого класса Employee. Ни один внешний метод не может читать эти поля или изменять их.

Поля экземпляра могут быть объявлены как public, однако делать этого не следует. Любые компоненты программы могут обращаться к общедоступным полям и модифицировать их содержимое. Это противоречит принципу инкапсуляции. Мы настоятельно рекомендуем всегда закрывать доступ к полям экземпляра с помощью ключевого слова private.

В заключение отметим, что два из трех полей экземпляра представляют собой объекты: поля name и hireDat являются ссылками на экземпляры классов String и Date. Это довольно распространенное явление: классы часто содержат экземпляры других классов.

Конструкторы

Рассмотрим конструктор класса Employee.

```
public Employee(String n, double s, int year, int month, int day)
{
   name = n;
   salary = s;
   GregorianCalendar calendar
        = new GregorianCalendar(year, month - 1, day);
   hireDay = calendar.getTime();
}
```

Имя конструктора совпадает с именем класса. Этот конструктор выполняется при создании объекта Employee, заполняя поля экземпляра заданными значениями.

Например, при создании экземпляра класса Employee с помощью выражения

```
new Employee("James Bond", 100000, 1950, 1, 1);
```

поля экземпляра будут заполнены такими значениями:

```
name = "James Bond";
salary = 100000;
hireDay = January 1, 1950;
```

Между конструкторами и другими методами есть существенная разница: конструктор можно вызывать только в сочетании с оператором new. Конструктор нельзя применить к существующему объекту, чтобы изменить информацию в его полях. Например, вызов

```
james.Employee("James Bond", 250000, 1950, 1, 1); // ОШИБКА
```

при компиляции приведет к ошибке.

Позднее в этой главе мы вернемся к конструкторам. Пока запомните следующее.

- Имя конструктора совпадает с именем класса.
- Класс может иметь несколько конструкторов.
- Конструктор может иметь один или несколько параметров либо не иметь их вовсе.
- Конструктор не возвращает никакого значения.
- Конструктор всегда вызывается совместно с оператором new.

Конструкторы в языках Java и C++ работают одинаково. Однако учтите, что все объекты в языке Java размещаются в динамической памяти и конструкторы вызываются только вместе с оператором new. Программисты, имеющие опыт работы на языке C++, часто допускают такую ошибку:

```
Employee number007("James Bond", 10000, 1950, 1, 1);
  // C++, а не Java
```

Это выражение в C++ работает, а в Java — нет.

Будьте осторожны и не называйте локальные переменные так же, как и поля экземпляра. *Например, приведенный ниже конструктор не сможет установить зарплату* сотрудника.

```
public Employee(String n, double s, ...)
  {
     String name = n;     // ОШИБКА
     double salary = s;   // ОШИБКА
  }
```

В конструкторе объявляются *локальные* переменные name и salary. Доступ к этим переменным возможен только внутри конструктора. Они *маскируют* поля экземпляра с теми же именами. Некоторые программисты — например, авторы этой книги — могут написать подобный код чисто автоматически. Подобные ошибки крайне трудно обнаружить. Нужно быть внимательным и не называть переменные именами полей.

Явные и неявные параметры

Методы объекта имеют доступ ко всем его полям. Рассмотрим следующий метод:

```
public void raiseSalary(double byPercent)
{
   double raise = salary * byPercent / 100;
   salary += raise;
}
```

Он устанавливает новое значение поля salary. (*Данный метод не возвращает никакого значения.*) Например, вызов метода

```
number007.raiseSalary(5);
```

увеличивает значение поля number007.salary в объекте number007 на 5%. Строго говоря, вызов данного метода приводит к выполнению двух выражений:

```
double raise = number007.salary * 5 /100;
number007.salary += raise;
```

Метод raiseSalary имеет два параметра. Первый параметр, называемый *неявным*, представляет собой объект типа Employee, который указывается перед именем метода. Второй параметр, число, указанное в скобках после имени метода, называется *явным*.

Легко видеть, что явные параметры перечисляются в объявлении метода, например double byPercent. Неявный параметр в объявлении метода не приводится.

В каждом методе ключевое слово this ссылается на неявный параметр. При желании можно переписать метод raiseSalary следующим образом:

```
public void raiseSalary(double byPercent)
{
   double raise = this.salary * byPercent / 100;
   this.salary += raise;
}
```

Некоторые программисты предпочитают такой стиль, поскольку в нем более четко различаются поля экземпляра и локальные переменные.

В языке C++ методы обычно определяются вне класса. Например:

```
void Employee::raiseSalary(double byPercent) // C++, а не Java
{
   ...
}
```

Если определить метод внутри класса, он автоматически станет подставляемым (inline).

```
class Employee
{
 ...
 int getName() { return name; } // Подставляемая функция в C++.
}
```

В языке Java все методы определяются внутри класса. Это не делает их подставляемыми. Виртуальная машина Java анализирует, как часто производится обращение к методу, и принимает решение о том, должен ли метод быть подставляемым.

Преимущества инкапсуляции

Рассмотрим довольно простые методы getName(), getSalary() и getHireDay().

```
public String getName()
{
   return name;
}

public double getSalary()
{
  return salary;
}

public Date getHireDay()
{
   return hireDay;
}
```

Они представляют собой яркий пример методов доступа (accessor). Поскольку они лишь возвращают значения полей экземпляра, иногда их называют *методами доступа к полю* (field accessor).

Но не проще ли было сделать поля name, salary и hireDay открытыми (т.е. объявить их как public) и не создавать отдельные методы доступа к ним?

Дело в том, что поле name доступно лишь для чтения. После того как значение этого поля будет установлено конструктором, ни один метод не сможет его изменить. Следовательно, у нас есть гарантия, что информация, хранящаяся в этом поле, не будет искажена.

Поле salary допускает запись, однако изменить его значение может только метод raiseSalary(). В частности, если окажется, что поле содержит неверное значение, нужно будет отладить только один метод. Если бы поле salary было открытым, причина ошибки могла бы находиться где угодно.

Иногда нужно иметь возможность читать и модифицировать содержимое поля. Для этого необходимо реализовать в составе класса следующие три элемента.

- Закрытое поле данных (ключевое слово private).
- Общедоступный метод доступа (ключевое слово public).
- Общедоступный модифицирующий метод (ключевое слово public).

Это намного утомительнее, чем просто сделать общедоступным одно-единственное поле данных, однако при этом программист получает существенные преимущества.

1. Внутреннюю реализацию класса можно изменять совершенно независимо от других классов.

   Предположим, например, что имя и фамилия сотрудника хранятся отдельно.

   ```
   String firstName;
   String lastName;
   ```

   Тогда метод getName должен формировать возвращаемое значение следующим образом:

   ```
   firstName + " " + lastName
   ```

   Остальные части программы остаются неизменными. Разумеется, и методы доступа, и модифицирующие методы должны быть переработаны, чтобы учесть новое представление данных.

2. Модифицирующие методы могут выполнять проверку ошибок, в то время как при присвоении полю некоторого значения ошибки не выявляются.

   Например, метод setSalary() может проверить, не стала ли зарплата отрицательной величиной.

Будьте осторожны при создании методов доступа, возвращающих ссылки на изменяемый объект. Создавая класс Emloyee, мы нарушили это правило: метод getHireDay() возвращает объект Date.

```
class Employee
{
   ...
   public Date getHire();
   {
      return hireDay;
```

```
    }
    ...
    private Date hireDay;
}
```

Это не соответствует принципу инкапсуляции! Рассмотрим пример неверного кода:

```
Employee harry = ...;
Date d = harry.getHireDay();
double tenYearsInMilliSeconds = 10 * 365.25 * 24 * 60 * 60 * 1000;
d.setTime(d.getTime() - (long)tenYearsInMilliSeconds);
  // Значение объекта изменено.
```

Причину ошибки трудно уловить. Обе ссылки `d` и `harry.hireDay` относятся к одному и тому же объекту (рис. 4.5). Применение модифицирующего метода к объекту `d` автоматически изменяет состояние объекта, содержащегося в классе `Employee`!

Чтобы вернуть ссылку на изменяемый объект, его нужно сначала *клонировать*. Клон — это точная копия объекта, расположенная в другом месте памяти. Детали клонирования рассмотрены в главе 6. Ниже приведен исправленный код.

```
class Employee
{
    ...
    public Date hetHireDay()
    {
        return (Date)hireDay.clone();
    }
    ...
}
```

Используйте метод `clone()`, если вам нужно скопировать изменяемое поле данных.

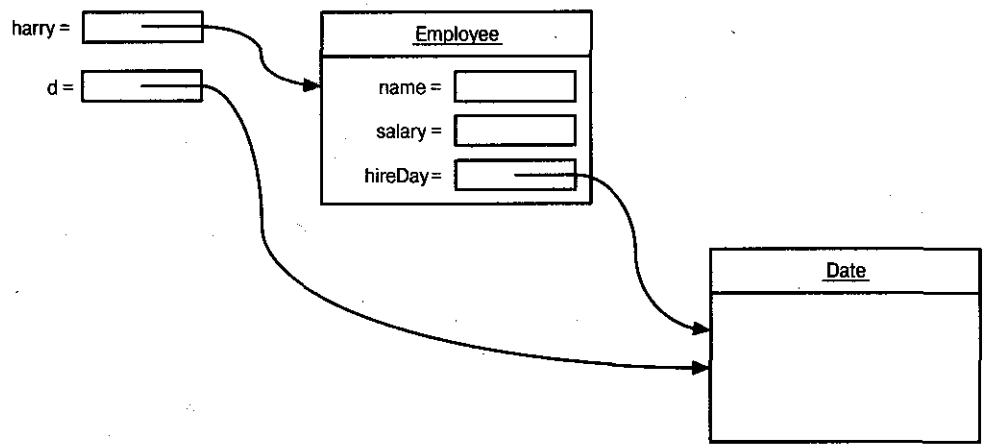


Рис. 4.5. Ссылки на изменяемый объект

Доступ к данным из различных экземпляров класса

Как вы уже знаете, метод имеет доступ к любым данным объекта, которому он принадлежит. Но он также может обращаться к закрытым данным *всех экземпляров своего класса*! Рассмотрим метод equals, сравнивающий между собой два экземпляра класса Employee.

```
class Employee
{
   ...
   boolean equals(Employee other)
   {
      return name.equals(other.name);
   }
}
```

Вызов этого метода выглядит следующим образом:

```
if (harry.equals(boss)) ...
```

Этот метод имеет доступ к закрытым полям объекта harry, что вовсе не удивительно. Но он также имеет доступ к полям объекта boss. Это вполне объяснимо, поскольку boss — объект Employee, а методы, принадлежащие классу Employee, могут обращаться к закрытым полям любого объекта этого типа.

В языке C++ действует такое же правило. Метод имеет доступ к переменным и функциям любого объекта своего класса.

Закрытые методы

При реализации класса мы сделали все поля данных закрытыми, поскольку предоставлять к ним доступ из других классов весьма рискованно. А как поступать с методами? Для взаимодействия с другими объектами необходимы открытые методы. Однако в ряде случаев для вычислений необходимы вспомогательные методы. Как правило, эти вспомогательные методы не являются частью интерфейса, поэтому указывать при их объявлении ключевое слово public нет необходимости; чаще всего они объявляются как private.

Чтобы сделать метод закрытым, измените ключевое слово public на private.

Сделав метод закрытым, совершенно не обязательно сообщать пользователям о его наличии. Использовать его будет только разработчик класса. Необходимо предоставлять информацию об открытых методах, поскольку другие части программы могут к ним обращаться.

Неизменяемые поля экземпляра

Поля экземпляра можно объявить с помощью ключевого слова final. Такое поле должно инициализироваться при создании объекта, после этого его значение изменить уже нельзя. Например, поле name класса Employee можно объявить неизменяемым, поскольку после создания объекта оно никогда не изменяется — метода setName не существует.

```
class Employee
{
   ...
   private final String name;
}
```

Модификатор final удобно применять при объявлении полей простых типов либо полей, типы которых задаются *неизменяемыми классами*. Неизменяемым называется класс, методы которого не позволяют изменить состояние объекта. Так, например, неизменяемым является класс String. Если класс допускает изменения, то ключевое слово final может стать источником недоразумений. Рассмотрим следующее выражение:

```
private final Date hiredate;
```

Оно означает, что переменная hiredate не изменяется после создания объекта. Однако это не означает, что состояние объекта, на который ссылается переменная, остается неизменным. В любой момент можно вызвать метод setTime().

Статические поля и методы

Во всех программах, которые мы до сих пор обсуждали, при объявлении метода main() использовался модификатор static. Рассмотрим действие этого модификатора.

Статические поля

Поле, имеющее модификатор static, существует в одном экземпляре. В то же время, если поле не статическое, то каждый объект содержит его копию. Предположим, что нам требуется присвоить уникальный идентификационный номер каждому сотруднику. Добавим в класс Employee поле id и статическое поле nextId.

```
class Employee
{
   ...
   private int id;
   private static int nextId = 1;
}
```

Теперь каждый объект Employee имеет свое поле id, кроме того, есть поле nextId, которое одновременно принадлежит всем экземплярам класса.

Попробуем объяснить это несколько иначе. Если существует тысяча объектов Employee, то в них есть тысяча полей id, по одному в каждом объекте. В то же время существует только один экземпляр статического поля nextId. Даже если не создан ни один объект Employee, статическое поле nextId существует. Оно принадлежит классу, а не конкретному объекту.

В большинстве объектно-ориентированных языков статические поля называются *полями класса* (class field). Термин "статический" унаследован от языка C++.

Реализуем простой метод.

```
public void setId()
{
   id = nextId;
   nextId++;
}
```

Допустим, нам нужно задать идентификационный номер объекта harry.

```
harry.setId();
```

Теперь значение поля id объекта harry задано, а значение статического поля nextId увеличено на единицу.

```
harry.id = ...;
Employee.nextId++;
```

Константы

Статические переменные используются довольно редко. В то же время статические константы используются гораздо чаще. Например, класс Math имеет статическую константу.

```
public class Math
{
   ...
   public static final double PI = 3.14159265358979323846;
   ...
}
```

Обратиться к этой константе в своей программе можно с помощью выражения Math.PI.

Если бы ключевое слово static было пропущено, константа PI была бы обычным полем экземпляра класса Math. Это значит, что для доступа к такой константе нужно было бы создать объект Math, причем каждый подобной объект имел бы свою копию константы PI.

Еще одна часто используемая статическая переменная — System.out. Она объявлена в классе System.

```
public class System
{
   ...
   public static final PrintStream out = ...;
   ...
}
```

Как мы уже несколько раз упоминали, *применять общедоступные поля не следует* никогда, поскольку любой объект сможет изменить их значения. Однако открытые константы (т.е. поля, объявленные с ключевым словом final) можно использовать смело. Поскольку поле out объявлено как final, ему нельзя присвоить другой поток вывода.

```
out = new PrintStream(...); // ОШИБКА -- поле out изменить нельзя.
```

Просмотрев код класса System, вы увидите метод setOut(), позволяющий присвоить полю System.out другой поток. Как же этот метод может изменить переменную, описанную как final? Дело в том, что метод setOut — *платформенно-ориентированный*, он реализован средствами, отличными от Java. Плетформенно-ориентированные методы могут обходить механизмы контроля, предусмотренные в языке Java. Это очень специфическое решение, которое ни в коем случае не следует повторять в своих программах.

Статические методы

Статические методы могут выполняться, даже если экземпляр класса не существует. Например, метод `pow()` из класса `Math` — статический. Выражение `Math.pow(x, y)` вычисляет x^y. При выполнении своей задачи этот метод не использует ни одного экземпляра класса `Math`. Иными словами, он не имеет неявного параметра. Это значит, что статические методы — это методы, в которых не используется объект `this`.

Поскольку статические методы не работают с объектами, с их помощью невозможно получить доступ к полям экземпляра. Однако статические методы имеют доступ к статическим полям класса. Ниже приведен пример статического метода.

```
public static int getNextId()
{
   return nextId; // Возвращает статическое поле
}
```

Чтобы вызвать этот метод, нужно указать имя класса.

```
int n = Employee.getNextId();
```

Можно ли пропустить ключевое слово `static` при описании этого метода? Да, но при этом для его вызова потребуется ссылка на объект типа `Employee`.

Для вызова статического метода можно использовать и объекты. Например, если `harry` — это объект `Employee`, то можно вместо `Employee.getNextId()` использовать вызов `harry.getNextId()`. Однако такое обозначение усложняет восприятие программы, поскольку для вычисления результата метод `getNextId()` не обращается к объекту `harry`. Мы рекомендуем для вызова статических методов использовать имена классов, а не объекты.

Статические методы следует применять в двух случаях.

- Когда методу не нужен доступ к информации о состоянии объекта, поскольку все необходимые параметры задаются явно (например, в методе `Math.pow`).
- Когда методу нужен доступ лишь к статическим полям класса (в качестве примера можно привести метод `Employee.getNextId()`).

Статические поля и методы в языках Java и C++, по существу, отличаются только синтаксически. В языке C++ для доступа к статическому полю или методу, находящемуся вне области видимости, можно использовать оператор `::`, например `Math::PI`.

Термин "статический" — исторический курьез. Сначала ключевое слово `static` было введено в языке C для обозначения локальных переменных, которые не уничтожались при выходе из блока. В этом контексте слово "статический" имеет смысл: переменная продолжает существовать после выхода из блока, а также при повторном входе в него. Затем слово "статический" в языке C приобрело второе значение — глобальные переменные и функции, к которым нельзя получить доступ из других файлов. Ключевое слово `static` было просто использовано повторно, чтобы не вводить новое. В итоге в языке C++ это ключевое слово было применено в третий раз, получив совершенно новую интерпретацию. Оно обозначает переменные и функции, принадлежащие классу, но не принадлежащие ни одному объекту этого класса. Именно этот смысл имеет ключевое слово `static` в языке Java.

Порождающие методы

Рассмотрим еще одно применение статических методов. Класс `NumberFormat` использует порождающие методы для создания объектов, соответствующих различным стилям форматирования.

```
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance()
NumberFormat percentFormatter = NumberFormat.getPercentInstance()
double x = 0.1;
System.out.println(currencyFormatter.format(x)); // Выводит $0.10
System.out.println(percentFormatter.format(x));  // Выводит 10%
```

Почему же не использовать для этой цели конструктор? Тому есть две причины.

- Конструктору нельзя присвоить произвольное имя. Его имя всегда должно совпадать с именем класса. В примере с классом `NumberFormat` имеет смысл применять разные имена для разных типов форматирования.
- При использовании конструктора тип объекта фиксирован. Если же применяются порождающие методы, они возвращают объект DecimalFormat, который наследует свойства `NumberFormat`. (Подробно вопросы наследования будут обсуждаться в главе 5.)

Метод main()

Заметим, что статические методы можно вызывать, даже если соответствующий объект еще не создан. Например, для того чтобы вызвать метод `Math.pow()`, объекты `Math` не нужны. По той же причине метод `main()` объявляется как статический.

```
public class Application
{
   public static void main(String[] args)
   {
     // Здесь создаются объекты.
   }
}
```

Каждый класс может содержать метод `main()`. Используя его, можно тестировать классы независимо друг от друга. Например, метод `main()` можно добавить в класс `Employee`.

```
class Employee
{
   public Employee(String n, double s,
      int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar =
         new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }
   . . .
   public static void main(String[] args) // Отладочный модуль.
```

```
        {
           Employee e = new Employee ("Romeo", 50000, 2003, 3, 31);
           e.raiseSalary(10);
           System.out.println(e.getName() + " " + e.getSalary());
        }
        . . .
     }
```

Если вы хотите протестировать только класс `Employee`, выполните следующую команду:

```
java Employee
```

Если класс `Employee` представляет собой часть большого приложения, запустите это приложение с помощью команды

```
java Application
```

и метод `main` класса `Employee` никогда не будет выполнен.

Программа, показанная в листинге 4.3, содержит версию класса `Employee` со статическим полем `nextId` и статическим методом `getNextId()`. Массив заполняется тремя объектами `Employee`, а затем выводится информация о сотрудниках. В заключение на экране отображается очередной доступный идентификационный номер.

Заметим, что класс `Employee` имеет статический метод `main()` для модульного тестирования. Попробуйте выполнить метод `main()` с помощью приведенных ниже команд.

```
java Employee
```

и

```
java StaticTest
```

Листинг 4.3. Содержимое файла `StaticTest.java`

```
public class StaticTest
{
   public static void main(String[] args)
   {
      // Заполнение массива staff объектами Employee.
      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Tom", 40000);
      staff[1] = new Employee("Dick", 60000);
      staff[2] = new Employee("Harry", 65000);

      // Вывод информации об объектах Employee.
      for (Employee e : staff)
      {
         e.setId();
         System.out.println("name=" + e.getName()
            + ",id=" + e.getId()
            + ",salary=" + e.getSalary());
      }

      int n = Employee.getNextId(); // Вызов статического метода.
```

```
      System.out.println("Next available id=" + n);
   }
}

class Employee
{
   public Employee(String n, double s)
   {
      name = n;
      salary = s;
      id = 0;
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public int getId()
   {
      return id;
   }

   public void setId()
   {
      id = nextId; // Установка идентификатора.
      nextId++;
   }

   public static int getNextId()
   {
      return nextId; // Возврат статического поля.
   }

   public static void main(String[] args) // Тестирование.
   {
      Employee e = new Employee("Harry", 50000);
      System.out.println(e.getName() + " " +
          e.getSalary());
   }

   private String name;
   private double salary;
   private int id;
   private static int nextId = 1;
}
```

Параметры методов

Рассмотрим термины, которые используются для описания способа передачи параметров методам (или функциям). Термин *вызов по значению* (call by value) означает, что метод получает значение, переданное ему вызывающим модулем. В противоположность этому *вызов по ссылке* (call by reference) означает, что метод получает от вызывающего модуля *адрес* (location) переменной. Таким образом, метод может *модифицировать* значение, хранящееся в переменной, переданной по ссылке, но не может изменять значение переменной, переданной по значению. Термин "вызов по..." относится к стандартной компьютерной терминологии, описывающей способ передачи параметров в различных языках программирования, а не только в языке Java. (На самом деле, существует еще и третий способ передачи параметров – *вызов по имени* (call by name), представляющий в основном исторический интерес, поскольку он был применен в языке Algol, одном из первых языков программирования высокого уровня.)

В языке Java всегда используется только вызов по значению. Это значит, что метод получает копии значений всех параметров. По этой причине метод не может модифицировать содержимое ни одного параметра, переданного ему.

Рассмотрим следующий вызов:

```
double percent = 10;
harry.raiseSalary(percent);
```

Не имеет значения, как именно реализован метод, мы знаем, что после его вызова значение переменной `percent` останется равным 10.

Изучим эту ситуацию подробнее. Допустим, что метод пытается утроить значение параметра.

```
public static void tripleValue(double x); // Не работает.
{
   x = 3 * x;
}
```

Вызовем этот метод.

```
double percent = 10;
tripleValue(percent);
```

Однако такой способ не работает. После вызова метода значение переменной `percent` остается равным 10. Происходит следующее.

1. Переменная `x` инициализируется копией значения параметра `percent` (т.е. числом 10).
2. Значение переменной `x` утраивается – теперь оно равно 30. Однако значение переменной `percent` остается равным 10 (рис. 4.6).
3. Метод завершает свою работу, и параметр `x` больше не используется.

Несмотря на то что после завершения метода передаваемые значения остаются неизменными, необходимо помнить, что существуют два типа параметров.

- Простые типы (числа, логические переменные и т.д.).
- Ссылки на объекты.

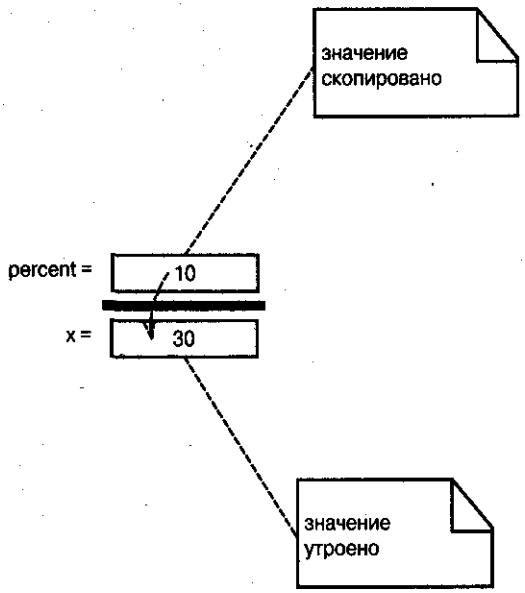


Рис. 4.6. Модификация значения внутри метода не оказывает действия на параметр, передаваемый из вызывающего метода

Как мы убедились, методы не могут модифицировать параметры простых типов. С объектами ситуация иная. Вы можете без труда реализовать метод, утраивающий зарплату сотрудников.

```
public static void tripleSalary(Employee x) // Работает
{
   x.raiseSalary(200);
}
```

При обработке следующего фрагмента кода будут выполнены перечисленные ниже действия.

```
harry = new Employee(...);
tripleSalary(harry);
```

1. Переменная `x` инициализируется копией значения переменной `harry`, т.е. ссылкой на объект.
2. Метод `raiseSalary()` применяется к этой ссылке на объект. Метод объекта `Employee`, на который указывают ссылки `x` и `harry`, увеличивает зарплату сотрудников на 200%.
3. Метод завершает свою работу, и параметр `x` больше не используется. Разумеется, переменная `harry` продолжает ссылаться на объект, в котором зарплата увеличена втрое (рис. 4.7).

Как видите, реализовать метод, изменяющий состояние объекта, передаваемого как параметр, довольно легко. Фактически такие изменения вносятся очень часто. Причина этого проста — метод получает копию ссылки на объект, поэтому и копия, и оригинал ссылки указывают на один и тот же объект.

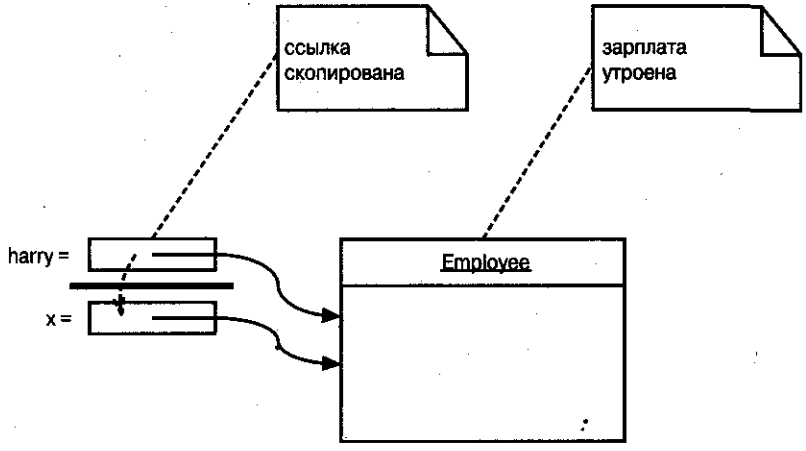


Рис. 4.7. Если параметр ссылается на объект, этот объект может быть модифицирован

Во многих языках программирования (в частности, в языках C++ и Pascal) есть два способа передачи параметров: вызов по значению и вызов по ссылке. Некоторые программисты (и, к сожалению, даже авторы некоторых книг) утверждают, что язык Java использует при передаче объектов только вызов по ссылке. Однако это не так. Попробуем написать метод, выполняющий обмен двух объектов `Employee`.

```
public static void swap(Employee x, Employee y) // Не работает.
{
   Employee temp = x;
   x = y;
   y = temp;
}
```

Если бы в языке Java для передачи объектов в качестве параметров использовался вызов по ссылке, этот метод работал бы так:

```
Employee a = new Employee("Alice", . . .);
Employee b = new Employee("Bob", . . .);
swap(a, b);
// Ссылается ли теперь a на Bob, а b — на Alice?
```

Однако на самом деле этот метод не меняет местами объектные ссылки, хранящиеся в переменных `a` и `b`. Параметры `x` и `y` метода `swap()` инициализируются *копиями* этих ссылок. Затем метод меняет местами эти копии.

```
// Ссылка x соответствует Alice, а ссылка y — Bob.
Employee temp = x;
x = y;
y = temp;
// Теперь x ссылается на Bob, а y — на Alice.
```

В конце концов, следует признать, что все было напрасно. По завершении работы метода переменные `x` и `y` уничтожаются, а исходные переменные `a` и `b` продолжают ссылаться на прежние объекты (рис. 4.8).

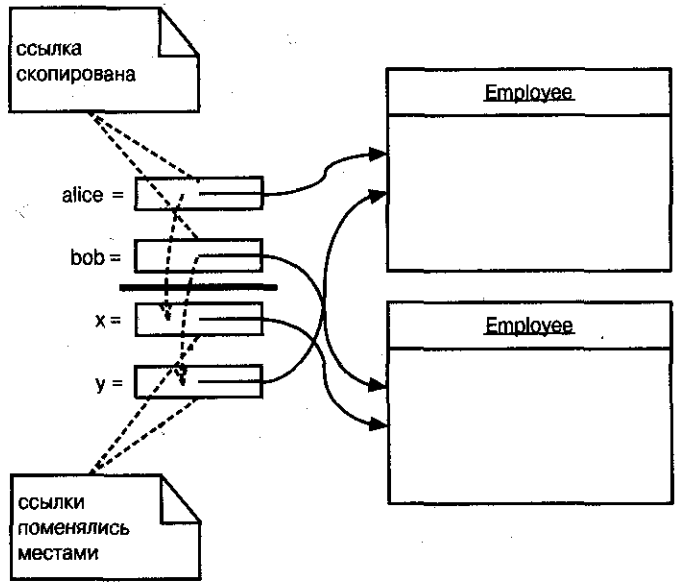


Рис. 4.8. Перемещение ссылок не влияет на значения в вызывающем методе

Таким образом, в языке Java для передачи объектов не используется вызов по ссылке. Вместо этого *ссылки на объекты передаются по значению.*

Ниже перечисляется, что может делать метод со своими параметрами, а что — нет.

- Метод не может изменять параметры простых типов (т.е. числа и логические значения).
- Метод может изменять *состояние* объекта, передаваемого как параметр.
- Метод не может изменять ссылки и переназначать их на новые объекты.

Программа, приведенная в листинге 4.4, иллюстрирует эти утверждения. Сначала программа безуспешно пытается утроить значение числового параметра.

```
Проверка метода tripleValue:
До: percent=10.0
В конце метода: x=30.0
После: percent=10.0
```

Затем программа успешно утраивает зарплату сотрудника.

```
Проверка метода tripleSalary:
До: salary=50000.0
В конце метода: salary=150000.0
После: salary=15000.0
```

После выполнения метода состояние объекта, на который ссылается переменная `harry`, изменяется. Ничего невероятного здесь нет, поскольку метод модифицировал состояние объекта с помощью копии ссылки на него.

В заключение программа демонстрирует безрезультатную работу метода swap().

```
Проверка метода swap:
До: a=Alice
До: b=Bob
В конце метода: x=Bob
В конце метода: y=Alice
После: a=Alice
После: b=Bob
```

Как видим, переменные x и y меняются местами, однако переменные a и b остаются прежними.

В языке C++ применяется как вызов по значению, так и вызов по ссылке. Например, легко можно реализовать методы `void tripleValue(double& x)` или `void swap(Employee& x, Emloyee& y)`. Эти методы модифицируют свои параметры, являющиеся ссылками.

Листинг 4.4. Содержимое файла ParamTest.java

```
public class ParamTest
{
   public static void main(String[] args)
   {
      /*
       Тест 1: методы не могут изменять числовые параметры.
      */
      System.out.println("Testing tripleValue:");
      double percent = 10;
      System.out.println("Before: percent=" + percent);
      tripleValue(percent);
      System.out.println("After: percent=" + percent);

      /*
         Тест 2: методы могут изменять состояние объектов,
         передаваемых в качестве параметров
      */
      System.out.println("\nTesting tripleSalary:");
      Employee harry = new Employee("Harry", 50000);
      System.out.println("Before: salary=" +
         harry.getSalary());
      tripleSalary(harry);
      System.out.println("After: salary=" +
          harry.getSalary());

      /*
         Тест 3: методы не могут приписывать новые объекты
           параметрам, являющимся ссылками на объекты.
      */
      System.out.println("\nTesting swap:");
      Employee a = new Employee("Alice", 70000);
      Employee b = new Employee("Bob", 60000);
      System.out.println("Before: a=" + a.getName());
      System.out.println("Before: b=" + b.getName());
      swap(a, b);
```

```
      System.out.println("After: a=" + a.getName());
      System.out.println("After: b=" + b.getName());
   }

   public static void tripleValue(double x) // Не работает.
   {
      x = 3 * x;
      System.out.println("End of method: x=" + x);
   }

   public static void tripleSalary(Employee x) // Работает.
   {
      x.raiseSalary(200);
      System.out.println("End of method: salary="
         + x.getSalary());
   }

   public static void swap(Employee x, Employee y)
   {
      Employee temp = x;
      x = y;
      y = temp;
      System.out.println("End of method: x=" +
          x.getName());
      System.out.println("End of method: y=" +
          y.getName());
   }
}

class Employee // Упрощенный класс Employee
{
   public Employee(String n, double s)
   {
      name = n;
      salary = s;
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   private String name;
   private double salary;
}
```

Формирование объектов

Мы уже знаем, как можно писать простые конструкторы, определяющие начальные состояния объектов. Однако создание объектов — очень важная задача, поэтому в языке Java предусмотрено много разнообразных механизмов для создания конструкторов. Эти механизмы рассматриваются ниже.

Перегрузка

Как вы помните, в классе `GregorianCalendar` предусмотрено несколько конструкторов. Создать объект этого класса позволяет выражение

```
GregorianCalendar today = new GregorianCalendar();
```

То же можно сделать с помощью следующей строки кода:

```
GregorianCalendar deadline
   = new GregorianCalendar(2099, Calendar, DECEMBER, 31);
```

Такая возможность называется *перегрузкой* (overloading). О перегрузке говорят в том случае, если несколько методов (в нашем случае несколько конструкторов) имеют одинаковое имя, но разные параметры. Компилятор должен сам определить, какой метод вызвать, сравнивая типы параметров, описанных в заголовках методов, с типами значений, указанных в вызове. Если ни один метод не соответствует вызову либо если одному вызову одновременно соответствует несколько вариантов, возникает ошибка компиляции. (Этот процесс называется *разрешением перегрузки* (overloading resolution).)

В языке Java можно перегрузить любой метод, а не только конструкторы. Таким образом, чтобы полностью описать метод, нужно указать его имя и типы параметров. Подобное написание называется *сигнатурой* метода (signature). Например, в классе String есть четыре общедоступных метода `indexOf()`. Они имеют следующие сигнатуры:

```
indexOf(int)
indexOf(int, int)
indexOf(String)
indexOf(String, int)
```

Тип возвращаемого методом значения не входит в его сигнатуру. Следовательно, нельзя создать два метода, имеющих одинаковые имена и типы параметров и отличающихся лишь типом возвращаемого значения.

Инициализация полей по умолчанию

Если значение поля в конструкторе явно не задано, ему автоматически присваивается значение по умолчанию: числам — нули, логическим переменным — значения `false`, а ссылкам на объект — `null`. Однако полагаться на действия по умолчанию *считается плохим стилем. Если поля инициализируются неявно, программа становится* менее понятной.

Между полями и локальными переменными есть существенная разница. Локальные переменные всегда должны явно инициализироваться в методе. Однако, если поле в составе класса не инициализируется явно, ему автоматически присваивается значение, задаваемое по умолчанию (0, значение `false` или ссылка `null`).

Рассмотрим в качестве примера класс Employee. Допустим, что в конструкторе значения некоторых полей не заданы. По умолчанию поле salary должно инициализироваться нулем, а поля name и hireDay — ссылкой null. Однако при вызове методов getName() или getHireDay() ссылка null может оказаться совершенно нежелательной.

```
Date h = harry.getHireDay();
calendar.setTime(h); // Если h = null, генерируется исключение.
```

Конструктор по умолчанию

Конструктор по умолчанию (default constructor) не имеет параметров. Например, конструктор по умолчанию класса Employee имеет следующий вид:

```
public Employee()
{
   name = "";
   salary = 0;
   hireDay = new Date();
}
```

Если в классе вовсе не определены конструкторы, вызывается конструктор по умолчанию. Этот конструктор присваивает *всем* полям экземпляра их значения, предусмотренные по умолчанию. Так, все числовые значения, содержащиеся в полях экземпляра, окажутся равными *нулю*, логические значения — false, а ссылки на объекты — null.

Если в классе есть хотя бы один конструктор и явно не определен конструктор без параметров, то при создании объектов необходимо задавать параметры конструктора. Например, класс Employee в листинге 4.2 имеет один конструктор:

```
Employee(String name, double salary, int y, int m, int d)
```

В этом случае нельзя создать объект, поля которого принимали бы значения по умолчанию. Другими словами, представленный ниже вызов приведет к возникновению ошибки.

```
e = new Employee();
```

Учтите, что конструктор по умолчанию вызывается, *только* если в классе не определены другие конструкторы. Если есть хотя бы один конструктор с параметрами и необходимо создавать экземпляр класса с помощью приведенного ниже выражения, следует явно определить конструктор без параметров.

```
new ИмяКласса()
```

Разумеется, если значения по умолчанию вас устраивают, можно создать следующий конструктор:

```
public ИмяКласса()
{
}
```

Явная инициализация полей

Поскольку конструкторы в классе можно перегружать, есть несколько способов задать начальное состояние полей его экземпляров. Всегда следует быть уверенным, что, независимо от вызова конструктора, все поля экземпляра имеют некие разумные значения.

Есть возможность присвоить каждому полю соответствующее значение в определении класса. Например:

```
class Employee
{
   ...
   private String name = "";
}
```

Это присваивание выполняется до вызова конструктора. Такой подход особенно полезен тогда, когда надо, чтобы, независимо от вызова конструктора класса, поле имело конкретное значение.

При инициализации поля не обязательно использовать константу. Ниже приведен пример, в котором поле инициализируется с помощью вызова метода.

```
class Employee
{
   static int assignId()
   { int r = nextId;
     nextId++;
     return r;
   }
   ...
   private int id = assignId();
}
```

В языке C++ нельзя инициализировать поля экземпляра непосредственно в описании класса. Значения всех полей должны задаваться в конструкторе. Однако в языке C++ есть синтаксическая конструкция, называемая *списком инициализации* (initializer list).

```
Employee::Employee(String n, double s,   int y, int m, int d) // C++
:  name(n),
   salary(s),
   hireDay(y, m, d)
{
}
```

В языке C++ эта синтаксическая конструкция используется для вызова конструкторов полей экземпляра. В языке Java поступать так нет никакой необходимости, поскольку объекты не могут содержать другие объекты; разрешается иметь только ссылки на них.

Имена параметров

Создавая даже самый тривиальный конструктор (а большинство из них таковыми и являются), трудно выбрать подходящие имена для его параметров.

Обычно в качестве имен параметров используются отдельные буквы.

```
public Employee(String n, double s)
{
   name = n;
   salary = s;
}
```

Однако недостаток такого подхода заключается в том, что, читая программу, невозможно понять, что означают параметры n и s.

Некоторые программисты добавляют к осмысленным именам параметров префикс "a".

```
public Employee(String aName, double aSalary)
{
   name = aName;
   salary = aSalary
}
```

Такая программа вполне понятна. Любой читатель может сразу определить, в чем заключается смысл параметра. Есть еще один широко распространенный прием. Чтобы использовать его, надо знать, что параметры *маскируют* (shadow) поля экземпляра с одинаковыми именами. Например, если вызвать метод с параметром salary, то эта переменная будет ссылаться на параметр, а не на поле экземпляра. Доступ к полю экземпляра осуществляется с помощью выражения this.salary. Напомним, что ключевое слово this означает неявный параметр, т.е. создаваемый объект. Следующий пример иллюстрирует данный подход:

```
public Employee(String name, double salary)
{
   this.name = name;
   this.salary = salary;
}
```

В языке C++ к именам полей экземпляра обычно добавляют префиксы, представляющие собой либо символ подчеркивания, либо букву (нередко для этой цели используется m или x). Например, поле, в котором записан размер зарплаты, может называться _salary, mSalary или xSalary. Разработчики, создающие программы на языке Java, как правило, так не поступают.

Вызов одного конструктора из другого

Ключевое слово this ссылается на неявный параметр метода. Однако у этого слова есть еще одно значение.

Если первый оператор конструктора имеет вид this(...), то вызывается другой конструктор этого же класса. Ниже приведен типичный пример.

```
public Employee(double s)
{
   // Вызывает конструктор Employee(String, double).
   this("Employee " + nextId, s);
   nextId++;
}
```

Если вы используете выражение new Employee(60000), то конструктор Employee(double) вызывает конструктор Employee(String, double).

Применять ключевое слово this для вызова другого конструктора очень полезно — при этом достаточно лишь один раз написать общий код, создающий объект.

Объект this в языке Java идентичен указателю this в языке C++. Однако в языке C++ невозможно вызвать один конструктор с помощью другого. Для того чтобы реализовать общий код инициализации объекта в языке C++, нужно создать отдельный метод.

Инициализационные блоки

Ранее мы рассмотрели два способа инициализации поля.

- Задать его значение в конструкторе.
- Присвоить значение при объявлении.

На самом деле в языке Java существует и *третий* механизм — использование *инициализационного блока* (initialization block). Такой блок выполняется каждый раз, когда создается объект данного класса. Рассмотрим следующий пример кода:

```
class Employee
{
   public Employee(String n, double s)
   {
      name = n;
      salary = s;
   }

   public Employee()
   {
      name = "";
      salary = 0;
   }
   ...

   private static int nextId;


   private int id;
   private String name;
   private double salary;
   ...

   // Инициализационный блок
   {
     id = nextId;
     nextId++;
   }
}
```

В этом примере начальное значение поля id задается в инициализационном блоке объекта, причем неважно, какой именно конструктор используется для создания экземпляра класса. Инициализационный блок выполняется первым, а тело конструктора — после него.

Этот механизм совершенно не обязателен и обычно не используется. Гораздо чаще применяются более понятные способы задания начальных значений полей.

В инициализационном блоке допустимо обращение в полям, определения которых находятся после данного блока. Некоторые версии Java-компилятора разработки Sun считают подобное обращение некорректным. Эта ошибка была зарегистрирована под номером # 4459133 и устранена в JDK 1.4.1. Несмотря на то что инициализация полей, определяемых после блока формально допустима, поступать так не рекомендуется во избежание возникновения циклических определений. Конкретные правила изложены в разделе 8.3.2.3 спецификации Java (`http://java.sun.com/docs/books/jls`). Поскольку правила достаточно сложны и учесть их в реализации компилятора крайне трудно, разработчикам рекомендуется помещать инициализационные блоки после определения полей.

Поскольку существует несколько способов инициализации полей, полезно подробно рассмотреть, какие действия выполняются при вызове конструктора.

1. Все поля инициализируются значениями, предусмотренными по умолчанию (0, `false` или `null`).
2. Инициализаторы всех полей и инициализационные блоки выполняются в порядке их следования в объявлении класса.
3. Если в первой строке конструктора вызывается другой конструктор, то выполняется вызванный конструктор.
4. Выполняется тело конструктора.

Каждый согласится, что код, отвечающий за инициализацию полей, нужно организовать так, чтобы в нем было легко разобраться. Например, было бы странным, если бы вызов конструкторов класса зависел от порядка объявления полей. Такой подход чреват ошибками.

Инициализировать статическое поле следует, либо задавая его начальное значение, либо используя *статический инициализационный блок* (static initialization block). Первый механизм мы уже рассматривали:

```
static int nextId = 1;
```

Если для инициализации статических полей класса нужен сложный код, удобно использовать статический инициализационный блок.

Разместите код внутри блока и пометьте его ключевым словом `static`. Пусть, например, нам нужно, чтобы идентификационные номера сотрудников начинались со случайного числа, не превышающего 10000.

```
// Статический инициализационный блок.
static
{
   Random generator = new Random();
   nextId = generator.nextInt(10000);
}
```

Статическая инициализация выполняется, когда класс загружается впервые. Аналогично полям экземпляра статические поля принимают значения 0, `false` или `null`, если программист не задал другие значения явным образом. Все операторы, задающие начальные значения статических полей, и статические блоки инициализации выполняются в порядке их перечисления в объявлении класса.

Оказывается, вполне возможно создать программу Hello, World, не использующую метод `main()`.

```
public class Hello
{
   static
   {
      System.out.println("Hello, World!");
   }
}
```

При выполнении команды `java Hello` класс загружается, статический блок инициализации выводит строку "`Hello, World!`", и лишь затем появляется сообщение о том, что метод `main()` не определен. Это сообщение можно подавить с помощью вызова метода `System.exit(0)` в конце статического инициализационного блока.

Программа, приведенная в листинге 4.5, иллюстрирует многие из вопросов, обсуждавшихся в этой главе.

- Перегрузка конструкторов.
- Вызов конструктора с помощью выражения `this(...)`.
- Использование конструктора по умолчанию.
- Использование инициализационного блока.
- Использование статического инициализационного блока.
- Инициализация поля экземпляра.

Листинг 4.5. Содержимое файла `ConstructorTest.java`

```
import java.util.*;

public class ConstructorTest
{
   public static void main(String[] args)
   {
    // Заполняет массив staff тремя объектами Employee.
      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Harry", 40000);
      staff[1] = new Employee(60000);
      staff[2] = new Employee();

      // Вывод информации о всех сотрудниках.
      for (Employee e : staff)
         System.out.println("name=" + e.getName()
            + ",id=" + e.getId()
            + ",salary=" + e.getSalary());
   }
}

class Employee
{
   // Три перегруженных конструктора.
   public Employee(String n, double s)
   {
```

```
      name = n;
      salary = s;
   }

   public Employee(double s)
   {
      // Вызов конструктора Employee(String, double).
      this("Employee #" + nextId, s);
   }

   // Конструктор по умолчанию.
   public Employee()
   {
    // Поле name инициализируется как ""-- см. ниже.
    // Поле salary задается неявно -- инициализируется нулем.
    // Поле id задается в блоке инициализиации.
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public int getId()
   {
      return id;
   }

   private static int nextId;

   private int id;
   private String name = ""; // Инициализация поля name.
   private double salary;

   // Статический блок инициализации.
   static
   {
      Random generator = new Random();
      // Задаем nextId как случайное число от 0 до 9999.
      nextId = generator.nextInt(10000);
   }

   // Инициализационный блок объекта
   {
      id = nextId;
      nextId++;
   }
}
```

API **java.util.Random 1.0**

- `Random()`

  Создает новый генератор случайных чисел.

- `int nextInt(int n)` 1.2

  Возвращает случайное число в интервале от 0 до `n-1`.

Уничтожение объекта и метод finalize()

В некоторых объектно-ориентированных языках, в частности в C++, есть явные деструкторы, предназначенные для уничтожения объектов. Основная их задача — освобождение памяти, занятой объектами. Поскольку в языке Java есть механизм автоматической сборки мусора, освобождать память вручную нет никакой необходимости, поэтому в языке Java деструкторы отсутствуют.

Разумеется, некоторые объекты используют, кроме памяти, и другие ресурсы, например файлы, или обрабатывают другие объекты, которые, в свою очередь, обращаются к системным ресурсам. В этом случае очень важно, чтобы ресурсы вовремя освобождались.

В любой класс можно добавить метод `finalize()`. Этот метод будет вызван перед тем, как система "сборки мусора" уничтожит объект. На практике, если вам нужно возобновить ресурсы и сразу использовать их повторно, *нельзя полагаться на метод* `finalize()`, — вы не знаете, когда именно этот метод будет вызван.

Существует метод `System.runFinalizerOnExit(true)`, гарантирующий, что метод `finalize()` будет вызван до того, как программа завершит свою работу. Однако этот метод крайне ненадежен и не рекомендован к использованию. В качестве альтернативы можно применить метод `Runtime.addShutdownHook()`. Дополнительную информацию о нем можно найти в документации по API.

Если ресурс должен быть освобожден сразу после его использования, нужно самостоятельно написать соответствующий код. Добавьте метод `dispose()` или `close()`, который нужно *явно* вызвать для очистки памяти. Если класс имеет метод `dispose()`, нужно вызвать данный метод по завершении работы с объектом этого класса.

Пакеты

Язык Java позволяет объединять классы в наборы, называемые *пакетами* (package). Пакеты облегчают организацию работы и позволяют отделить классы, созданные одним разработчиком, от классов, разработанных другими программистами.

Стандартная библиотека языка Java содержит большое количество пакетов; в качестве примеров можно привести `java.lang`, `java.util`, `java.net` и др. Стандартные пакеты языка Java представляют собой иерархические структуры. Подобно каталогам на диске компьютера, пакеты могут быть вложены один в другой. Все стандартные пакеты принадлежат иерархиям `java` и `javax`.

В основном пакеты используются для обеспечения уникальности имен классов. Предположим, двух программистов осенила блестящая идея создать класс `Employee`. Если оба класса будут находиться в разных пакетах, конфликт не возникнет. Чтобы обеспечить абсолютную уникальность имени пакета, компания Sun рекомендует использовать доменное имя компании в Internet (которое по определению уникально),

записанное в обратном порядке. В составе пакета можно создавать подпакеты и использовать их в разных проектах. Например, один из авторов этой книги имеет зарегистрированный домен `horstmann.com`. Записав это имя в обратном порядке, можно использовать его как название пакета — `com.horstmann`. В дальнейшем в этом пакете можно создать подпакет, например `com.horstmann.corejava`.

Единственная цель вложенных пакетов — гарантия уникальности имен. С точки зрения компилятора между вложенными пакетами нет никакой связи. Например, пакеты `java.util` и `java.util.jar` никак не связаны друг с другом. Каждый из них представляет собой независимую коллекцию классов.

Импортирование классов

Класс может использовать все классы из собственного пакета и все *общедоступные* классы из других пакетов.

Доступ к классам из других пакетов можно получить двумя путями. Во-первых, можно указывать полное имя пакета перед именем *каждого* класса. Например:

```
java.util.Date today = new java.util.Date();
```

Очевидно, что этот способ слишком утомителен. Более простой и распространенный способ — использовать ключевое слово `import`. В этом случае имя пакета перед именем класса записывать не обязательно.

Импортировать можно как один конкретный класс, так и весь пакет. Операторы `import` следует поместить в начало исходного файла (после всех операторов `package`). Например, все классы из пакета `java.util` можно импортировать с помощью выражения

```
import java.util.*;
```

После этого в строке кода, указанной ниже, имя пакета не указывается.

```
Date today = new Date();
```

Можно также импортировать отдельный класс из пакета.

```
import java.util.Date;
```

Импортировать все классы проще. На размер кода это не влияет. Однако, если явным образом указать импортируемый класс, становится ясно, какие именно классы будут использованы в программе.

Работая с Eclipse, можно использовать пункт меню Source⇨Organize Imports. В результате выражения типа `import java.util.*;` будут автоматически преобразованы в последовательности строк, предназначенных для импортирования отдельных классов.

```
import java.util.ArrayList;
import java.util.Date;
```

Эта возможность несколько упрощает работу программиста.

Однако следует заметить, что оператор `import` со звездочкой можно применять для импортирования только одного пакета. Нельзя использовать обозначение `import java.*` или `import java.*.*`, чтобы импортировать все пакеты, имена которых содержат префикс `java`.

В большинстве случаев импортируется весь пакет, независимо от его размера. Единственный вариант, при котором на пакет следует обратить особое внимание, —

конфликт имен. Например, и пакет `java.util`, и пакет `java.sql` содержат класс `Date`. Допустим, вы пишете программу, импортирующую оба этих пакета.

```
import java.util.*;
import java.sql.*;
```

Если теперь попытаться использовать класс `Date`*, возникнет ошибка компиляции.*

```
Date today; // ОШИБКА -- java.util.Date или java.sql.date?
```

Компилятор не может определить, какой класс `Date` вам нужен. Решить эту проблему можно, добавив конкретный оператор `import`.

```
import java.util.*;
import java.sql.*;
import java.util.Date;
```

А если вам на самом деле нужны оба класса `Date`? В этом случае нужно указывать полное имя пакета для каждого имени класса.

```
java.util.Date deadline = new java.util.Date();
java.sql.Date today = new java.sql.Date();
```

Обнаружение классов в пакетах является задачей *компилятора*. В ссылках, находящихся в файлах классов, всегда содержатся полные имена пакетов.

Программисты, использующие язык C++, считают, что оператор `import` *является аналогом директивы* `#include`. Однако между ними нет ничего общего. В языке C++ директиву `#include` нужно применять, поскольку компилятор C++ не просматривает файлы, за исключением того, который он компилирует, и файлов, указанных в директиве `#include`. Компилятор языка Java "видит" содержимое всех файлов при условии, если он "знает", где их искать.

В языке Java можно вовсе не применять механизм импортирования, явно назвав все пакеты, например `java.util.Date`. В языке C++ избежать использования директивы `#include` нельзя.

Единственное преимущество оператора `import` — удобство. Он позволяет использовать более короткие имена классов, не указывая полное имя пакета. Например, после оператора `import java.util.*` (или `import java.util.Date`) на класс `java.util.Date` можно ссылаться по имени `Date`.

Аналогичный механизм работы с пакетами в языке C++ реализован в виде директивы `namespace`. Ключевые слова `package` и `import` в языке Java можно считать аналогами директив `namespace` и `using` в C++.

Импортирование статических методов и полей

В JDK 5.0 выражение `import` *было расширено. Теперь вы можете импортировать* не только классы, но также статические методы и поля. Предположим, например, что в начале файла с исходным текстом вы разместили следующую строку кода:

```
import static java.lang.System.*;
```

Это позволит использовать статические методы и поля, определенные в классе System, не указывая имени класса.

```
out.println("Goodbye, World!"); // вместо System.out
exit(0);                        // вместо System.exit
```

Вы также можете явным образом импортировать статические методы или поля.

```
import static java.lang.System.out;
```

Вряд ли можно встретить программиста, который захочет сокращать такие выражения, как `System.out` или `System.exit`. Полученный в результате код станет более трудным для восприятия. Однако новая возможность окажется полезной как *минимум в двух случаях*.

1. При использовании математических функций. Если вы импортируете статические методы класса `Math`, то сможете непосредственно задавать в программе имена математических функций. Согласитесь, что выражение

   ```
   sqrt(pow(x, 2) + pow(y, 2))
   ```

   выглядит гораздо естественнее, чем следующая строка кода:

   ```
   Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))
   ```

2. При использовании констант. Если в программе применяется большое количество констант, возможность импортирования статических полей придется как нельзя более кстати. Выражение

   ```
   if (d.get(DAY_OF_WEEK) == MONDAY)
   ```

   проще для восприятия, чем строка

   ```
   if (d.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY)
   ```

Добавление классов в пакеты

Чтобы поместить класс в пакет, нужно указать имя пакета в начале исходного файла *перед* определением класса. Например, файл `Employee.java` в листинге 4.7 начинается следующими строками:

```
package com.horstmann.corejava;

public class Employee
{
   ...
}
```

Если оператор `package` в исходном файле не указан, то классы, описанные в этом файле, помещаются в *пакет по умолчанию* (default package). Пакет по умолчанию не имеет имени. До сих пор все рассмотренные нами классы принадлежали пакету по умолчанию.

Пакеты следует помещать в подкаталог, путь к которому соответствует полному имени пакета. Например, все файлы классов в пакете `com.horstmann.corejava` должны находиться в подкаталоге `com/horstmann/corejava` (*соответственно* `com\horstmann\corejava` в системе Windows).

Программы, приведенные в листингах 4.6 и 4.7, распределены по двум пакетам: класс `PackageTest` принадлежит пакету по умолчанию, а класс `Employee` — пакету `com.horstmann.corejava`. Следовательно, файл `Employee.class` должен содержаться в подкаталоге `com/horstmann/corejava`. Другими словами, структура каталогов должна выглядеть следующим образом:

```
.(базовый каталог)
  !-- PackageTest.java
  !-- PackageTest.class
  !-- com/
       !-- horstmann/
            !-- corejava/
                 !-- Employee.java
                 !-- Employee.class
```

Чтобы скомпилировать эту программу, перейдите в каталог, содержащий файл `PackageTest.java`, *и выполните следующую команду:*

```
javac Package.java
```

Компилятор автоматически найдет файл `com/horstmann/corejava/Employee.java` и скомпилирует его.

Рассмотрим ситуацию, встречающуюся на практике чаще. В данном примере пакет по умолчанию не используется, вместо этого классы распределены по разным пакетам (`com.horstmann.corejava` и `com.mycompany`).

```
. (базовый каталог)
    !-- com/
         !--horstmann/
         !    !-- corejava/
         !         !--Employee.java
         !         !--Employee.class
         !-- mycompany/
              !-- PayrollApp.java
              !-- PayrollApp.class
```

Компиляция осуществляется из *базового каталога*, т.е. того каталога, в котором содержится подкаталог `com`.

```
javac com/mycompany/PayrollApp.java
java com.mycompany.PayrollApp
```

Помните, что компилятор работает с файлами (при указании файла задается путь и расширение `.java`), а интерпретатор оперирует с классами (для класса указывается пакет).

Компилятор не проверяет структуру каталогов. Предположим, например, что исходный файл начинается со следующей директивы:

```
package com.mycompany;
```

Этот файл можно скомпилировать, даже если он не находится в каталоге `com/mycompany`. Исходный файл будет скомпилирован без ошибок, однако при попытке выполнить эту программу виртуальная машина не найдет классы, полученные в результате компиляции.

Листинг 4.6. Содержимое файла `PackageTest.java`

```
import com.horstmann.corejava.*;
   // В этом пакете определен класс Employee.

import static java.lang.System.*;
```

```
public class PackageTest
{
   public static void main(String[] args)
   {
      // Поскольку использован оператор import, имя
      // com.horstmann.corejava.Employee использовать
      // необязательно
      Employee harry = new Employee("Harry Hacker", 50000,
         1989, 10, 1);

      // Увеличение зарплаты на 5%.
      harry.raiseSalary(5);

      // Вывод информации о сотруднике
      // (при этом используется объект java.lang.System.out)
      out.println("name=" + harry.getName()
         + ",salary=" + harry.getSalary());
   }
}
```

Листинг 4.7. Содержимое файла `Employee.java`

```
package com.horstmann.corejava;
   // Классы в этом файле являются частью пакета.

import java.util.*;
   // Операторы import помещаются после оператора package

public class Employee
{
   public Employee(String n, double s,
      int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar
         = new GregorianCalendar(year, month - 1, day);
      // В классе GregorianCalendar январю соответствует 0.
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }
```

```
   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   private String name;
   private double salary;
   private Date hireDay;
}
```

Как виртуальная машина определяет, где находятся классы

Как вы уже знаете, классы хранятся в подкаталогах файловой системы. Путь к классу должен совпадать с именем пакета. Кроме того, чтобы поместить классы в *архив*, можно использовать утилиту `jar`. В одном архивном файле содержится много файлов классов и подкаталогов, что позволяет экономить память и сокращать время доступа к ним. (Файлы JAR более детально обсуждаются в главе 10.)

Например, тысячи классов рабочей библиотеки содержатся в файле `rt.jar`. Этот файл можно найти в подкаталоге `jre/lib` набора инструментальных средств JDK.

Для организации файлов и подкаталогов в файлах JAR используется формат ZIP. Обрабатывать `rt.jar` и другие JAR-файлы можно с помощью любой утилиты, поддерживающей ZIP.

В примерах, рассмотренных ранее, каталог `com/horstmann/corejava` представлял собой подкаталог каталога, в которых размещались программы. Однако такое расположение является не очень удобным. Как правило, к файлам пакета должны обращаться различные программы. Чтобы обеспечить им совместный доступ к пакетам, необходимо сделать следующее.

1. Поместить классы в один или несколько специальных каталогов, скажем `/home/user/classdir`. Заметим, что этот каталог является *базовым* по отношению к дереву пакета. Если потребуется добавить класс `com.horstmann.corejava.Employee`, то файл класса следует поместить в подкаталог `/home/user/classdir/com/horstmann/corejava`.
2. Установить *путь к классу*. Путь к классу — это набор всех базовых каталогов, которые могут содержать файлы классов.

Установка пути к классу зависит от среды, в которой компилируется программа. Если используется пакет JDK, есть две возможности: указывать при вызове компилятора и интерпретатора опцию `-classpath` или задать значение переменной окружения `CLASSPATH`.

Детали зависят от операционной среды. В среде Unix элементы пути к классу отделяются друг от друга двоеточиями.

```
/home/user/classdir:.:/home/user/archives/archive.jar
```

В среде Windows они разделяются точками с запятой.

```
c:\classes;.;c:\archives\archive.jar
```

В обоих случаях точка означает текущий каталог.
Путь к классу содержит:

- имя базового каталога `/home/user/classdir` или `c:\classes`;
- обозначение текущего каталога (`.`);
- имя файла JAR `/home/user/archives/archive.jar` или `c:\archives\archive.jar`.

Поиск классов всегда выполняется в файлах рабочей библиотеки (`rt.jar` и других JAR-файлах в каталогах `jrelib` и `jre/lib/ext`). Имена этих файлов явно указывать не обязательно.

В версиях 1.0 и 1.1 Java Development Kit системные классы хранились в файле `classes.zip`, *который указывался в составе пути к классам.*

Рассмотрим, как устанавливается путь в классам с помощью опции компилятора.

```
javac -classpath /home/user/classdir:.:/home/user/archives
    archive.jar MyProg.java
```

(Вся команда должна быть введена в одной строке. В среде Windows для отделения элементов пути друг от друга используются точки с запятой.)

Вместо `-classpath` можно использовать опцию `-cp`. Однако в версиях, предшествующих JDK 5.0, опция `-cp` распознавалась только интерпретатором `java`. При работе с компилятором `javac` необходимо было использовать `-classpath`.

В составе пути перечисляются все каталоги и архивные файлы, рассматриваемые при поиске классов как начальные позиции. Рассмотрим простой путь:

```
/home/user/classdir:.:/home/user/archives/archive.jar
```

Предположим, что интерпретатор выполняет поиск файла, содержащего класс `com.horstmann.corejava.Employee`. Сначала он ищет его в системных файлах, которые хранятся в архивах, находящихся в каталогах `jre/lib` и `jre/lib/ext`. В этих файлах искомого класса нет, поэтому интерпретатор анализирует пути. Он ищет следующие файлы:

- `/home/user/classdir/com/horstmann/corejava/Employee.class`.
- `com.horstmann/corejava/Employee.class`, начиная с текущего каталога.
- `com.horstmann/corejava/Employee.class` в каталоге `/home/user/archives/archive.jar`.

Компилятор `javac` всегда ищет файлы в текущем каталоге, а интерпретатор `java` обращается к текущему каталогу, только если в пути к классам указана точка ".". Если путь к классам не указан, никаких проблем не возникает — по умолчанию в него включается текущий каталог ".". Но если вы задали путь и забыли указать точку, то *ваша программа будет скомпилирована без ошибок, однако выполниться не сможет.*

Как задать путь к классам

Путь к классам можно задать с помощью опции `-classpath`, указываемой при вызове программ `javac` и `java`. Однако такой способ не очень удобен. Можно воспользоваться альтернативной возможностью и задать значение переменной окружения `CLASSPATH`. Ниже указано, как это сделать в операционной среде Unix/Linux и Windows.

- В среде Unix/Linux нужно отредактировать файл, в котором хранятся параметры оболочки.

  Если используется оболочка C, добавьте в файл `.cshrc`, находящийся в рабочем каталоге, *следующую строку*:

  ```
  setnv CLASSPATH /home/user/classdir:
  ```

  Если используется оболочка Bourne Again, или `bash`, добавьте в файл `.bashrc` или `.bash_profile`, находящийся в рабочем каталоге, строку

  ```
  export CLASSPATH=/home/user/classdir:
  ```

- Работая с Windows 95/98/Me, отредактируйте файл `autoexec.bat` на устройстве, с которого осуществляется загрузка (чаще всего это диск C:). Включите в файл строку

  ```
  SET CLASSPATH=c:\user\classdir;
  ```

  Убедитесь, что слева и справа от символа = нет пробелов.

- В среде Windows NT/2000/XP откройте панель управления. Затем выберите пиктограмму System и вкладку Environment. В поле Variable задайте слово `CLASSPATH`. В поле ввода значений укажите требуемый путь, например `c:\user\classdir;` (рис. 4.9). Если переменная `CLASSPATH` существует, достаточно отредактировать ее значение.

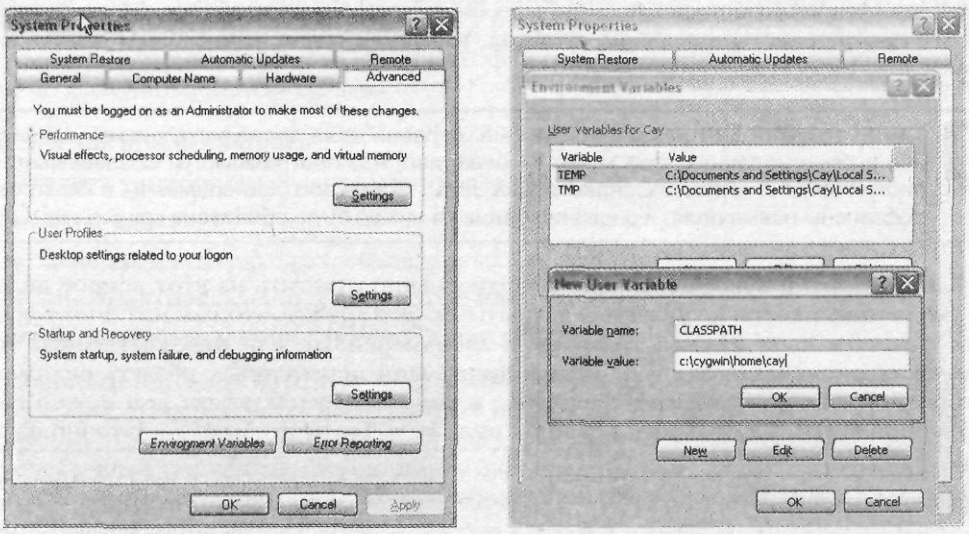


Рис. 4.9. Установка пути к классам в Windows XP

Область видимости пакета

Мы уже встречались с модификаторами доступа `public` и `private`. Элементы, помеченные ключевым словом `public`, могут использоваться любым классом. Закрытые элементы, т.е. те, при определении которых использовалось ключевое слово `private`, могут использоваться только тем классом, в котором они были определены. Если ни один модификатор доступа не указан, то элемент (класс, метод или переменная) доступен всем методам в том же самом *пакете*.

Вспомним программу, приведенную в листинге 4.2. Класс `Employee` не определен как общедоступный. Следовательно, любой другой класс из того же самого пакета — в данном случае пакета, заданного по умолчанию, — например класс `EmployeeTest`, может получить доступ к нему. Для классов такой подход надо признать вполне разумным. Однако для переменных этот способ доступа неудачен. Переменные должны быть явно помечены как `private`, или по умолчанию их область видимости будет расширена до пределов пакета. Это, конечно, нарушает принцип инкапсуляции. В процессе работы над программой разработчики часто забывают указать ключевое слово `private`. Вот пример из класса `Window`, принадлежащего пакету `java.awt`, который поставляется с пакетом JDK.

```
public class Window extends Container
{
   String warningString;
   ...
}
```

Заметим, что переменная `warningString` не имеет модификатора доступа `private`! Это значит, что методы всех классов в пакете `java.awt` могут обращаться к ней и изменить ее значение (например, присвоить ей строку `Trust me!`). Фактически все методы, имеющие доступ к `warningString`, принадлежат классу `Window`, поэтому эту переменную смело можно было бы объявить как закрытую. Мы подозреваем, что программисты, писавшие этот код, торопились и просто забыли указать ключевое слово `private`.

Как ни странно, этот недостаток не был устранен даже после того, как мы указали на нее в семи изданиях этой книги, — очевидно, что программисты, создававшие эту библиотеку, не читают "Основы языка Java". Более того, со временем в класс были добавлены новые поля, и снова половина из них не была объявлена как `private`.

А может, это и вовсе не проблема? Однозначно ответить на этот вопрос нельзя. По умолчанию пакеты не являются закрытыми. Это значит, что каждый может добавлять в пакеты свои классы. Разумеется, злонамеренные или невежественные программисты могут написать код, модифицирующий переменные, область видимости которых ограничена пакетом. Например, в ранних версиях языка Java можно было легко проникнуть в другой класс пакета `java.awt`. Для этого было достаточно начать свой класс со строки

```
package java.awt;
```

Затем нужно было поместить полученный файл класса в подкаталог `java\awt`, и доступ к содержимому пакета `java.awt` открыт. Таким образом можно было изменить строку предупреждения, отображаемую на экране (рис. 4.10).

В версии 1.2 создатели пакета JDK запретили загрузку классов, определенных пользователем, если их имя начиналось с `java.`! Разумеется, эта защита не распространяется на ваши собственные классы. Вместо этого можно использовать другой механизм, называемый *герметизацией пакета* (package sealing), который ограничивает доступ к пакету. Выполнив герметизацию пакета, вы запрещаете добавлять к нему классы. В главе 10 показано, как создать JAR-файл, содержащий герметичные пакеты.

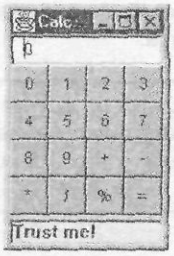


Рис. 4.10. Изменение строки в окне аплета

Комментарии и документирование

Пакет JDK содержит полезный инструмент под названием `javadoc`, генерирующий на базе исходных файлов документацию в формате HTML. Фактически описание API является результатом применения утилиты `javadoc` к исходному коду стандартной библиотеки языка Java.

Добавив к своему исходному коду комментарии, начинающиеся последовательностью символов `/**`, легко создать документацию, имеющую профессиональный вид. Это очень удобный способ, поскольку он позволяет совместно хранить как код, так и документацию к нему. Если документацию поместить в отдельный файл, то со временем она перестанет соответствовать коду. В то же время, поскольку комментарии являются неотъемлемой частью исходного файла, документацию легко обновить, повторно выполнив утилиту `javadoc`.

Включение комментариев

Утилита `javadoc` извлекает информацию о следующих элементах.

- Пакетах.
- Классах и интерфейсах, объявленных как `public`.
- Методах, объявленных как `public` или `protected`.
- Полях, объявленных как `public` или `protected`.

Защищенные элементы, для объявления которых используется ключевое слово `protected`, будут рассмотрены в главе 5, а интерфейсы – в главе 6.

Разрабатывая программу, можно (и даже необходимо) комментировать каждый из перечисленных элементов. Комментарии помещаются непосредственно *перед* элементом, к которому они относятся. Комментарии начинаются с разделителя `/**` и заканчиваются символами `*/`.

Комментарии, имеющие вид `/** ... */`, содержат произвольный текст, содержащий *дескриптор* (tag). Дескриптор начинается символом `@`, например `@author` или `@param`.

Первое предложение текста комментариев должно представлять собой *краткое описание* (summary statement). Утилита `javadoc` автоматически генерирует страницы, состоящие из кратких описаний.

В самом тексте можно использовать элементы языка HTML, например, `<em>...</em>` — для выделения текста курсивом, `<code>...</code>` для установки моноширинного шрифта, `<strong>...</strong>` — для выделения текста полужирным шрифтом и даже `<img ...>` — для вставки рисунков. Следует, однако, избегать заголовков (`<h1>` — `<h6>`) и горизонтальных линий (`<hr>`), поскольку они могут повлиять на форматирование документа.

Если комментарии содержат ссылки на другие файлы, например рисунки, поместите эти файлы в подкаталоги, начинающиеся словом `doc`. Утилита `javadoc` скопирует эти каталоги и файлы, содержащиеся в них, из исходного каталога в каталог для документации.

Комментарии к классу

Комментарии к классу должны помещаться *после* директив `import`, непосредственно перед определением класса. Ниже приведен пример подобных комментариев.

```
/**
   Объект класса <code> Card</code> имитирует игральную карту,
   например червонную даму. Карта имеет масть и ранг
   (1=туз, 2...10, 11=валет, 12=дама, 13=король).
*/
public class Card
{
   ...
}
```

Многие программисты начинают каждую строку документации звездочкой. Например:

```
/**
 *  Объект класса <code> Card</code> имитирует игральную карту,
 *  например червонную даму. Карта имеет масть и ранг
 *  (1=туз, 2...10, 11=валет,    12=дама, 13=король).
 */
```

Мы так не поступаем, поскольку при этом процесс обновления комментариев затрудняется. Никто не хочет заново расставлять звездочки после того, как порядок строк окажется нарушенным. Однако некоторые текстовые редакторы помогают выполнять эту тяжелую работу. Если вы уверены, что все потенциальные пользователи вашей программы будут применять именно такие текстовые редакторы, можете украсить ваши комментарии звездочками.

Комментарии к методам

Комментарии должны непосредственно предшествовать методу, который они описывают. Кроме дескрипторов общего назначения можно использовать специальные дескрипторы.

`@param` *переменная описание*

Этот дескриптор добавляет в описание метода раздел "parameters" ("параметры"). Раздел параметров можно "растянуть" на несколько строк. Кроме того, можно использовать элементы языка HTML. Все дескрипторы `@param`, относящиеся к одному методу, должны быть сгруппированы вместе.

`@return` описание

Данный дескриптор добавляет в описание метода раздел "returns" ("возвращаемое значение"). Этот раздел также может занимать несколько строк. Допустимы также дескрипторы языка HTML.

`@throws` *описание_класса*

Этот дескриптор означает, что метод может генерировать исключение. Исключения будут рассмотрены в главе 11.

Рассмотрим пример комментариев к методу.

```
/**
   Увеличивает зарплату сотрудников.
   @param Переменная byPercent содержит величину
     в процентах, на которую повышается зарплата
     (например, 10 = 10%).
   @return Размер повышения
*/
public double raiseSalary(double byPercent)
{
   double raise = salary &* byPercent / 100;
   salary += raise;
   return raise;
}
```

Комментарии к полям

Документировать нужно лишь общедоступные поля — обычно они представляют собой статические константы. Например,

```
/**
   Масть черви.
*/
public static final int HEARTS = 1;
```

Комментарии общего характера

В комментариях к классу можно использовать следующие дескрипторы:

`@author` *имя*

Этот дескриптор создает раздел "author" ("автор"). В комментариях может быть несколько таких дескрипторов — по одному на каждого автора.

`@version` *текст*

Этот дескриптор создает раздел "version" ("версия"). В данном случае "*текст*" означает произвольное описание текущей версии.

При создании любых комментариев, предназначенных для создания документации, можно использовать также следующие дескрипторы:

`@since` *текст*

Этот дескриптор создает раздел "since" ("начиная с..."). Слово "текст" означает описание версии, в которой впервые было реализовано данное свойство. Например, `@since version 1.7.1`.

`@deprecated` *текст*

Этот дескриптор добавляет сообщение о том, что класс, метод или переменная не рекомендованы к применению. Подразумевается, что после дескриптора `@deprecated` следует некоторое выражение. Например:

```
@deprecated Use <code>setVisible(true)</code> instead
```

С помощью дескрипторов `@see` и `@link` можно использовать гипертекстовые ссылки на соответствующие внешние документы или на позицию в составе того же документа.

`@see` *ссылка*

Этот дескриптор добавляет ссылку в раздел "See also" ("См. также"). Его можно использовать в комментариях как к классам, так и к методам. Здесь под словом "*ссылка*" подразумевается одна из следующих конструкций.

- *пакет.класс#элемент метка*
- `<a href="...">`*метка*`</a>`
- "*текст*"

Первый вариант встречается чаще всего. Здесь нужно указать имена класса, метода или переменной, а утилита `javadoc` вставит в документацию соответствующую гипертекстовую ссылку. Например, приведенная ниже команда создает ссылку на метод `raiseSalary(double)` класса `com.horstmann.corejava.Employee`.

```
@see com.horstmann.corejava.Employee#raiseSalary(double)
```

Заметьте, что здесь для отделения имени класса от имени метода или переменной нужно использовать символ #, а не точку. Компилятор `javac` корректно обрабатывает точки, выступающие в качестве разделителей между именами пакетов, подпакетов, классов, внутренних классов, методов и переменных. Однако утилита `javadoc` не настолько тщательно проработана, поэтому возникает необходимость в специальном синтаксисе.

Если вслед за дескриптором `@see` стоит символ <, то программист должен указать гипертекстовую ссылку. Ссылаться можно на любой URL-адрес. Например:

```
@see <a href="www.horstmann.com/corejava.html">Web-страница книги
   "Основы языка Java"</a>
```

В каждом их этих случаев можно указать необязательную *метку*, которая играет роль якоря для ссылки. Если метку не указать, то якорем ссылки будет считаться имя кода или URL.

Если вслед за дескриптором `@see` стоит символ ", то текст отображается в разделе "see also". Например:

```
@see "Основы языка Java, том 2"
```

Для одного и того же элемента можно использовать несколько дескрипторов `@see`, однако их следует сгруппировать вместе.

При желании можно поместить гипертекстовые ссылки на другие классы или методы в любом комментарии. Для этого нужно включить в требуемую позицию специальный дескриптор вида {link *пакет.класс#элемент метка*}. Описание элемента подчиняется тем же правилам, что и дескриптор @see.

Комментарии к пакетам и обзоры

Комментарии к классам, методам и переменным помещаются обычно непосредственно в исходных файлах. При этом комментарии выделяются символами /** ... */. Однако для генерации комментариев к пакету в каждый каталог, где находится пакет, нужно добавить файл package.html. Весь текст, содержащийся между дескрипторами <body> ... </body>, извлекается утилитой javadoc.

Кроме того, все исходные файлы можно сопроводить обзорными комментариями. Они размещаются в файле под названием overview.html, размещенном в родительском каталоге, содержащем все исходные файлы. Весь текст, находящийся между дескрипторами <body> ... </body>, извлекается утилитой javadoc. Эти комментарии отображаются на экране, когда пользователь выбирает Overview.

Извлечение комментариев

Будем считать, что docDirectory — это имя каталога, в котором должны храниться HTML-файлы. Выполним следующие действия.

1. Перейдем в каталог, содержащий исходные файлы, подлежащие документированию. Если документированию подлежат вложенные пакеты, например com.horstmann.corejava, нужно перейти в каталог, содержащий подкаталог com. (Именно в этом каталоге должен храниться файл overview.html.)
2. Для документирования одного пакета выполним команду

   ```
   javadoc -d docDirectory имя_пакета
   ```

   Для документирования нескольких пакетов выполним команду

   ```
   javadoc -d docDirectory имя_пакета_1 имя_пакета_2 ...
   ```

   Если файлы находятся в пакете, заданном по умолчанию, вместо приведенных выше команд нужно выполнить следующую:

   ```
   javadoc -d docDirectory *.java
   ```

Если пропустить опцию -d docDirectory, HTML-файлы будут извлечены в текущий каталог, что вызовет путаницу; мы не рекомендуем делать это.

При вызове javadoc можно указывать различные опции. Например, чтобы включить в документацию дескрипторы @author и @version, можно использовать опции -author и -version. Опция -link позволяет включать ссылки на стандартные классы. Например, приведенная ниже команда автоматически создаст ссылку на документацию, находящуюся на Web-узле Sun.

```
javadoc -link http://java.sun.com/j2se/5.0/docs/api *.java
```

Другие опции описаны в документации на javadoc, которую можно найти по адресу http://java.sun.com/products/jdk/javadoc/index.html.

Если вам надо выполнить более тонкую настройку, например, создать документацию в формате, отличающемся от HTML, можно разработать свой собственный *доклет* (doclet), позволяющий генерировать документацию произвольного вида. Более подробную информацию о данном подходе можно получить на Web-странице `http://java.sun.com/j2se/javadoc`.

При работе над программами может оказаться полезным доклет DocCheck (`http://java.sun.com/j2se/javadoc/doccheck/`). Он *просматривает* исходные файлы и отмечает пропущенные комментарии.

Рекомендации по разработке классов

Завершим эту главу некоторыми советами, которые облегчат вам объектно-ориентированную разработку классов.

1. *Всегда храните данные в переменных, объявленных как* `private`.

   Первое и главное требование: всеми средствами избегайте нарушения инкапсуляции. Иногда приходится писать методы доступа к полю или модифицирующие методы, но предоставлять доступ к полям не следует. Горький опыт показал, что способ представления данных может изменяться, однако порядок их использования изменяется гораздо реже. Если данные закрыты, их представление не влияет на использующий класс, поэтому ошибки искать легче.

2. *Всегда инициализируйте данные.*

   Язык Java не инициализирует локальные переменные, однако поля в объектах инициализируются. Не полагайтесь на действия по умолчанию, инициализируйте переменные явным образом с помощью конструкторов.

3. *Не используйте в классе слишком много простых типов.*

   Несколько *связанных* между собой полей простых типов следует объединять в новый класс. Такие классы проще для понимания, и их легче модифицировать. Например, следующие четыре поля в классе `Customer` надо объединить в новый класс `Address`:

   ```
   private String street;
   private String city;
   private String state;
   private int zip;
   ```

   Представленный таким образом адрес легче изменять.

4. *Не для всех полей надо создавать методы доступа и модификации.*

   Очевидно, что при выполнении программы надо получать сведения о зарплате сотрудника, кроме того, ее приходится время от времени изменять. Однако есть поля, которые после создания объекта вовсе не изменяются. К ним относится массив аббревиатур названий штатов США в классе `Address`.

5. *Используйте стандартную форму определения класса.*

   Содержимое класса всегда перечисляется в следующем порядке:

   - общедоступные элементы;

- элементы, область видимости которых ограничена пакетом;
- закрытые элементы.

Внутри каждого раздела перечисляются:

- *методы экземпляра*;
- статические методы;
- поля экземпляра;
- статические поля.

Такой порядок вполне оправдан. Пользователей вашего класса больше интересует открытый интерфейс, чем детали закрытой реализации. Кроме того, их больше интересуют методы, чем данные.

Однако в настоящее время общего мнения об оптимальном стиле программирования еще нет. Согласно стилю программирования на языке Java, принятом в компании Sun, рекомендуется сначала перечислять поля, а затем — методы. Независимо от того, какого стиля вы придерживаетесь, главное — не отступать от него.

6. *Разбивайте на части слишком большие классы.*

   Этот совет, конечно, слишком общий: то, что кажется "слишком большим" одному программисту, кажется нормальным другому. Однако, если есть очевидная возможность разделить один сложный класс на два класса попроще, воспользуйтесь ею. (Только опасайтесь другой крайности. Вряд ли оправданы десять классов, в каждом из которых есть только один метод.)

   Ниже приведен пример плохого стиля.

```
class CardDeck // Плохой стиль.
{
   public CardDeck() { ... }
   public void shuffle() { ... }
   public int getTopValue() { ... }
   public int getTopSuit() { ... }
   public void draw() { ... }

   private int[] value;
   private int[] suit;
```

   На самом деле в этом классе реализованы две разные концепции: во-первых, *карточный стол* с методами `shuffle()` (тасование) и `draw()` (раздача) и, во-вторых, *игральная карта* с методами для проверки ранга и масти карты. Разумнее было бы ввести класс `Card`, представляющий собой отдельную игральную карту. Тогда мы имели бы два класса, каждый со своей задачей.

```
class CardDeck
{
   public CardDeck() { ... }
   public void shuffle() { ... }
   public int getTopValue() { ... }
   public int getTopSuit() { ... }
   public void draw() { ... }
```

```
   private Card[] cards;
}

class Card
{
   public Card (int aValue, int aSuit) { ... }
   public int getValue() { ... }
   public int getSuit() { ... }

   private int[] value;
   private int[] suit;
}
```

7. *Давайте классам и методам осмысленные имена.*

   Классы, как и переменные, следует называть именами, отражающими их смысл. (В стандартной библиотеке есть, примеры, для которых данное правило нарушается, например, класс `Date` описывает время, а не дату.)

   Удобно принять следующее соглашение: имя класса должно быть именем существительным (`Order`) или именем существительным, которому предшествует имя прилагательное (`RushOrder`), или герундий (слово, заканчивающееся на "-ing", например `BillingAddress`). Как правило, методы доступа должны начинаться словом `get`, представленным строчными буквами (`getSalary`), а модифицирующие методы – словом `set`, также представленным строчными буквами (`setSalary`).

Глава 5

Наследование

В этой главе...

- Классы, суперклассы и подклассы
- Object: глобальный суперкласс
- Отражение
- Нумерованные типы
- Как применять наследование

В главе 4 мы изучали классы и объекты. Данная глава посвящена *наследованию* — фундаментальному понятию объектно-ориентированного программирования. Идея, лежащая в основе наследования, заключается в том, что новые классы можно создавать из уже существующих. При наследовании методы и поля существующего класса используются повторно (наследуются) вновь создаваемым классом, причем для адаптации нового класса к новым ситуациям в него добавляют дополнительные поля и методы. Этот прием играет в языке Java весьма важную роль.

Если вы имеете опыт работы лишь с такими процедурно-ориентированными языками, как C, Visual Basic или COBOL, тщательно изучите эту главу. Для программистов, работавших на языке C++ или других объектно-ориентированных языках, например языке Smaltalk, сведения, изложенные в этой главе, будут знакомы. Следует, однако, учесть, что механизмы наследования, реализованные в языке Java и в других объектно-ориентированных языках, значительно отличаются друг от друга.

В конце главы мы изучим механизм *отражения*, позволяющий исследовать свойства классов в ходе выполнения программы. Отражение — мощный, но очень сложный механизм. Поскольку отражение больше интересует разработчиков инструментальных средств, чем прикладных программистов, при первом чтении можно ограничиться беглым просмотром этого раздела.

Классы, суперклассы и подклассы

Вернемся к классу Employee, рассмотренному в предыдущей главе. Допустим, что вы работаете в компании, в которой работа менеджеров учитывается иначе, чем работа остальных сотрудников. Разумеется, менеджеры во многих отношениях являются обычными наемными сотрудниками. Однако сотрудникам за их работу выплачивается заработная плата, в то время как менеджеры за свои достижения получают еще и премии. В этой ситуации следует применять наследование. Почему? Потому что нужно определить новый класс Manager, содержащий ряд методов и полей. Однако некоторые из этих методов и полей уже определены в классе Employee, причем *все* поля этого класса подходят и для класса Manager. Говоря более абстрактно, между классами Manager и Employee существует очевидное отношение "является" ("is-a"). Каждый менеджер *является* сотрудником: это отношение "является" — признак наследования.

Ниже приводится пример определения класса Manager, порожденного из класса Employee. Для обозначения наследования в языке Java используется ключевое слово extends.

```
class Manager extends Employee
{
   Дополнительные методы и поля
}
```

Механизмы наследования в языках Java и C++ похожи. В языке Java вместо символов :: используется ключевое слово extends. Любое наследование в языке Java является открытым, причем аналога закрытому и защищенному наследованию, существующему в языке C++, в языке Java нет.

Ключевое слово extends означает, что на базе существующего создается новый класс. Существующий класс называется *суперклассом*, *базовым* или *родительским*, а вновь создаваемый — *подклассом*, *производным* или *дочерним*. Среди программистов, использующих язык Java, наиболее широко распространены термины "суперкласс" и "подкласс", хотя некоторые специалисты предпочитают пользоваться понятиями "родительский/дочерний", более тесно связанными со словом "наследование".

Класс Employee является суперклассом. Это не значит, что он имеет превосходство над своим подклассом или обладает более широкими функциональными возможностями. *На самом деле все наоборот*: подкласс *шире*, чем суперкласс. Например, анализируя код класса Manager, легко увидеть, что он инкапсулирует больше данных и содержит больше методов, чем его суперкласс Employee.

Префиксы *супер-* и *под-* пришли в программирование из теории множеств. Множество всех сотрудников содержит в себе множество всех менеджеров. В этом случае говорят, что множество сотрудников является *супермножеством* (superset) по отношению к множеству менеджеров. Иначе говоря, множество всех менеджеров является *подмножеством* (subset) множества всех сотрудников.

Класс Manager имеет новое поле, в котором хранится размер премии, и новый метод, позволяющий задавать эту величину.

```
class Manager extends Employee
{
   ...
   public void setBonus(double b)
   {
      bonus = b;
   }
   private double bonus;
}
```

В этих методах и полях нет ничего особенного. Имея объект `Manager`, можно просто применять метод `setBonus()`.

```
Manager boss = ...;
boss.setBonus(5000);
```

Разумеется, с помощью объекта `Employee` метод `setBonus()` вызвать невозможно — его нет среди методов, определенных в этом классе. Однако методы `getName()` и `getHireDay()` можно вызывать, пользуясь объектами `Manager`, поскольку они наследуются от суперкласса `Employee`.

Поля `name`, `salary` и `hireDay` также наследуются от суперкласса. Каждый объект `Manager` имеет четыре поля: `name`, `salary`, `hireDay` и `bonus`. Определяя подкласс, *нужно указать лишь отличия между подклассом и суперклассом. Разрабатывая классы*, следует помещать методы общего назначения в суперкласс, а более специальные методы — в подкласс. Такое выделение функциональных свойств общего назначения *путем их размещения в суперклассе широко распространено в объектно-ориентированном* программировании.

Однако некоторые методы суперкласса не подходят для подкласса `Manager`. В частности, метод `getSalary` должен возвращать сумму базовой зарплаты и премии. Следовательно, нужно *переопределить* метод, т.е. реализовать новый метод, *замещающий* метод суперкласса.

```
class Manager extends Employee
{
   ...
   public void getSalary()
   {
      ...
   }
   ...
}
```

Как реализовать этот метод? На первый взгляд это просто — нужно лишь вернуть сумму полей `salary` и `bonus`.

```
public void getSalary()
   {
      return salary + bonus; // Не работает.
   }
```

Однако выясняется, что этот способ не работает. Метод `getSalary()` класса `Manager` *не имеет доступа к закрытым полям суперкласса.* Другими словами, метод `getSalary()` класса `Manager` не может непосредственно обратиться к `salary`, несмотря на то, что каждый объект `Manager` имеет поле с таким именем. Только методы класса `Employee` имеют доступ к закрытым полям суперкласса. Если методам класса `Manager` нужен доступ к закрытым полям, они должны сделать то же, что и

любой другой метод, – использовать открытый интерфейс, в данном случае – метод `getSalary()` класса `Employee`.

Итак, попробуем все сначала. Вместо непосредственного обращения к полю `salary` нужно вызвать метод `getSalary()`.

```
public void getSalary()
   {
      double baseSalary = getSalary(); // Все еще не работает.
      return baseSalary + bonus;
   }
```

Проблема заключается в том, что метод `getSalary()` вызывает сам себя, поскольку в классе `Manager` есть свой метод с таким именем (именно его мы пытаемся реализовать). Возникает бесконечная цепочка вызовов одного и того же метода, что приводит к аварийному завершению программы.

Нужно явно указать, что мы хотим вызвать метод `getSalary()` из суперкласса `Employee`, а не из текущего класса. Для этой цели используется специальное ключевое слово `super`. Приведенное ниже выражение вызывает метод `getSalary()` именно из класса `Employee`.

```
super.getSalary()
```

Вот как выглядит правильный вариант метода `getSalary()` для класса `Manager`.

```
public void getSalary()
   {
      double baseSalary = super.getSalary();
      return baseSalary + bonus;
   }
```

Некоторые программисты считают ключевое слово `super` аналогом ссылки `this`. Однако эта аналогия не вполне точна — ключевое слово `super` не означает ссылку на объект. Например, его значение нельзя присвоить другой объектной переменной. Это слово лишь сообщает компилятору, что нужно вызвать метод суперкласса.

Как видим, подкласс может *добавлять* поля, а также *добавлять* и *переопределять* методы суперкласса, причем при наследовании ни одно поле или метод из класса не удаляется.

Для вызова метода суперкласса в языке Java используется ключевое слово `super`. В языке C++ для этого можно указывать имя суперкласса вместе с оператором разрешения области видимости `::`. Например, метод `getSalary()` класса `Manager` в языке C++ можно было бы вызвать с помощью выражения `Employee::getSalary()`.

В заключение напишем конструктор класса.

```
public Manager(String n, double s, int year, int month, int day)
{
   super(n, s, year, month, day);
   bonus = 0;
}
```

Здесь ключевое слово super имеет другой смысл. Приведенное ниже выражение означает вызов конструктора суперкласса Employee с параметрами n, s, year, month и day.

```
super(n, s, year, month, day);
```

Поскольку конструктор класса Manager не имеет доступа к закрытым полям класса Employee, он должен инициализировать их, вызывая другой конструктор с помощью ключевого слова super. Вызов, содержащий обращение super, должен быть первым оператором в конструкторе подкласса.

Если конструктор подкласса не вызывает явно ни один конструктор суперкласса, то автоматически вызывается конструктор по умолчанию, принадлежащий данному суперклассу. Если в суперклассе нет конструктора по умолчанию, а конструктор подкласса не вызывает явно другой конструктор суперкласса, то компилятор Java отобразит сообщение об ошибке.

Напомним, что ключевое слово this применяется в двух случаях: для указания ссылки на неявный параметр и для вызова другого конструктора того же класса. Аналогично ключевое слово super используется в двух ситуациях: для вызова метода суперкласса и конструктора суперкласса. При вызове конструкторов ключевые слова this и super имеют почти одинаковый смысл. Вызов конструктора должен быть первым оператором в другом конструкторе. Параметры конструктора передаются либо другому конструктору того же класса (this), либо конструктору суперкласса (super).

В языке C++ вызов конструктора super не применяется. Вместо этого для создания суперкласса используется список инициализации. В языке C++ конструктор класса Manager выглядел бы следующим образом:

```
Manager::Manager(String n, double s, int year, int month, int day) // C++
: Employee(n, s, year, MONTH, DAY)
{
   bonus = 0;
}
```

После переопределения метода getSalary() для объектов Manager все менеджеры получат премии.

Рассмотрим конкретный пример: создадим объект Manager и установим размер премии.

```
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
boss.setBonus(5000);
```

Образуем массив, в котором должны храниться три объекта Employee.

```
Employee[] staff = new Employee[3];
```

Заполним его объектами Manager и Employee.

```
staff[0] = boss;
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

Выведем зарплату каждого сотрудника.

```
for (Employee e : staff)
   System.out.println(e.getName() + " " + e.getSalary());
```

В результате выполнения цикла отображаются следующие строки:

```
Carl Cracker 85000.0
Harry Hacker 50000.0
Tommy Tester 40000.0
```

Поскольку `staff[1]` и `staff[2]` являются объектами `Employee`, они выводят базовые зарплаты соответствующих сотрудников. Однако объект `staff[0]` относится к классу `Manager`, и его метод `getSalary()` добавляет к базовой зарплате премию.

Обратите особое внимание на вызов

```
e.getSalary()
```

Здесь вызывается именно тот метод `getSalary()`, который необходим в данном случае. Заметим, что объявленным типом переменной `e` является `Employee`, однако *фактическим* типом объекта, на который может ссылаться переменная `e`, может быть как класс `Employee` (когда `i=1` или 2), так и класс `Manager` (когда `i=0`).

Когда переменная `e` ссылается на объект `Employee`, выражение `e.getSalary()` вызывает метод `getSalary()` класса `Employee`. Однако, если переменная `e` ссылается на объект `Manager`, вызывается метод `getSalary()` класса `Manager`. Виртуальной машине известен фактический тип объекта, на который ссылается переменная `e`, поэтому с вызовом метода проблемы не возникают.

Способность переменной (например, `e`) ссылаться на объекты, имеющие разные фактические типы, называется *полиморфизмом*. Автоматический выбор нужного метода во время выполнения программы называется *динамическим связыванием* (dynamic binding). Соответствующие вопросы будут обсуждаться далее в этой главе.

В языке Java нет необходимости объявлять виртуальный метод. Динамическое связывание выполняется по умолчанию. Если вы хотите, чтобы метод невозможно было переопределить, пометьте его ключевым словом `final` (оно будет рассматриваться ниже).

В листинге 5.1 представлен код, демонстрирующий подсчет заработной платы с использованием объектов `Employee` и `Manager`.

Листинг 5.1. Содержимое файла `ManagerTest.java`

```
import java.util.*;

public class ManagerTest
{
   public static void main(String[] args)
   {
      // Создаем объект Manager.
    Manager boss = new Manager("Carl Cracker",
       80000, 1987, 12, 15);
    boss.setBonus(5000);

    Employee[] staff = new Employee[3];

     // Заполнение массива staff объектами Manager и Employee
```

```
      staff[0] = boss;
      staff[1] = new Employee("Harry Hacker", 50000,
         1989, 10, 1);
      staff[2] = new Employee("Tommy Tester", 40000,
         1990, 3, 15);

      // Вывод информации обо всех объектах Employee
      for (Employee e : staff)
         System.out.println("name=" + e.getName()
            + ",salary=" + e.getSalary());
   }
}

class Employee
{
   public Employee(String n, double s,
      int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar
         = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
  {
      return salary;
   }

    public Date getHireDay()
    {
       return hireDay;
    }

    public void raiseSalary(double byPercent)
    {
       double raise = salary * byPercent / 100;
       salary += raise;
   }

   private String name;
   private double salary;
   private Date hireDay;
}

class Manager extends Employee
{
   /**
      @param n Имя сотрудника
```

```
      @param s Зарплата
      @param year Год найма
      @param month Месяц найма
      @param day День найма
   */
   public Manager(String n, double s,
      int year, int month, int day)
   {
      super(n, s, year, month, day);
      bonus = 0;
   }

   public double getSalary()
   {
      double baseSalary = super.getSalary();
      return baseSalary + bonus;
   }

   public void setBonus(double b)
   {
      bonus = b;
   }

   private double bonus;
}
```

Иерархия наследования

Наследование не обязательно ограничивается одним уровнем классов. Например, на основе класса `Manager` можно создать подкласс `Executive`. Совокупность всех классов, производных от общего суперкласса, называется *иерархией наследования* (inheritance hierarchy). Условно иерархия наследования показана на рис. 5.1. Путь от конкретного класса к его потомкам в иерархии называется *цепочкой наследования* (inheritance chain).

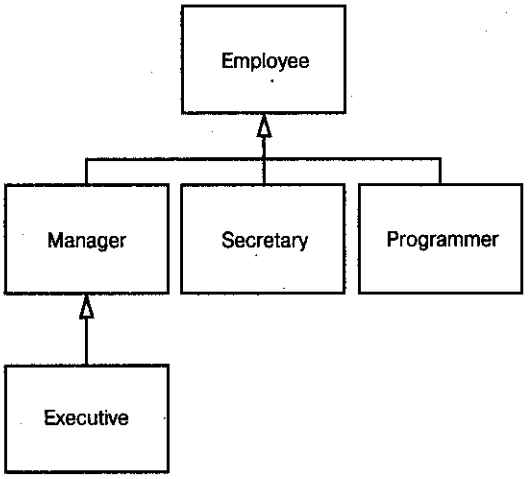


Рис. 5.1. Иерархия наследования для класса `Employee`

Обычно для класса существует несколько цепочек наследования. На основе класса `Employee` можно создать подкласс `Programmer` или класс `Secretary`, причем они будут совершенно независимыми как от класса `Manager`, так и друг от друга. Процесс формирования подклассов можно продолжать сколь угодно долго.

В языке Java множественное наследование не поддерживается. Задачи, для которых в других языках применяется множественное наследование, в Java решаются с помощью механизма интерфейсов (интерфейсы будут рассматриваться в следующей главе).

Полиморфизм

Существует простое правило, позволяющее определить, стоит ли в конкретной ситуации применять наследование или нет. Если между объектами существует отношение "является" ("is-a"), то каждый объект подкласса является объектом суперкласса. Например, каждый менеджер является сотрудником. Следовательно, имеет смысл сделать класс `Manager` подклассом класса `Employee`. Естественно, обратное утверждение неверно — не каждый сотрудник является менеджером.

Другой способ — *принцип подстановки* (substitution principle). Этот принцип гласит, что объект подкласса можно использовать вместо любого объекта суперкласса.

Например, объект подкласса можно присвоить переменной суперкласса.

```
Employee e;
e = new Employee(...); // Объект класса Employee.
e = new Manager(...);  // Можно и так.
```

В языке Java объектные переменные являются *полиморфными* (polymorphic). Переменная типа `Employee` может ссылаться как на объект `Employee`, так и на объект любого подкласса класса `Employee` (например, `Manager`, `Executive`, `Secretary` и т.п.).

Применение этого принципа продемонстрировано в листинге 5.1.

```
Manager boss = new manager(...);
Employee[] staff = new Employee[3];
staff[0] = boss;
```

Здесь переменные `staff[0]` и `boss` ссылаются на один и тот же объект. Однако переменная `staff[0]` рассматривается компилятором только как объект `Employee`. Это значит, что допускается следующий вызов:

```
boss.setBonus(5000); // OK
```

В то же время выражение, приведенное ниже, некорректно.

```
staff[0].setBonus(5000); // ОШИБКА
```

Дело в том, что переменная `staff[0]` объявлена как объект `Employee`, а метода `setBonus()` в этом классе нет.

Присвоить ссылку на объект суперкласса переменной подкласса невозможно. Например, следующий оператор является недопустимым:

```
Manager m = staff[i]; // ОШИБКА
```

Причина очевидна: не все сотрудники являются менеджерами. Если бы это присваивание оказалось выполненным и переменная `m` могла бы ссылаться на объект

Employee, который менеджером не является, то впоследствии оказался бы возможным вызов метода m.setBonus(...), что привело бы к ошибке при выполнении программы.

В языке Java массив ссылок на объекты подкласса можно преобразовать в массив ссылок *на объекты суперкласса. При этом выполнять явным образом приведение типов* не требуется. В качестве примера рассмотрим массив ссылок на объекты Manager:

```
Manager[] managers = new Manager[10];
```

Вполне допустимо преобразовать его в массив Employee[].

```
Employee[] staff = managers; // OK
```

Действительно, каждый менеджер является сотрудником, аналогично объект Manager содержит все поля и методы, присутствующие в объекте Employee. Однако тот факт, что переменные managers и staff ссылаются на один и тот же массив, может вызвать неприятные последствия. Рассмотрим следующее выражение:

```
staff[0] = new Employee("Harry Hacker", ...);
```

Компилятор допускает подобное присваивание. Но staff[0] и manager[0] — это одна и та же ссылка, поэтому ситуация выглядит так, как будто мы пытаемся присвоить ссылку на объект, описывающий рядового сотрудника, переменной, соответствующей объекту менеджера. В результате формально становится возможным вызов managers[0].setBonus(1000), который повлечет обращение к несуществующему методу и разрушение содержимого памяти.

Для того чтобы предотвратить подобную ситуацию, в созданном массиве запоминается тип элементов и отслеживается допустимость хранящихся ссылок. Например, если массив создан с помощью выражения new Manager[10], то он рассматривается как массив элементов Manager и попытка записать в него ссылку на объект Employee приведет к возникновению исключения ArrayStoreException.

Динамическое связывание

Рассмотрим подробно вызов метода, принадлежащего некоторому объекту. При этом выполняются следующие действия.

1. Компилятор проверяет объявленный тип объекта и имя метода. Допустим, происходит вызов метода x.f(args), причем неявный параметр x объявлен как экземпляр класса C. Заметим, что может существовать несколько методов с именем f, имеющих разные типы параметров (например, f(int) и метод f(String)). Компилятор нумерует все методы с именем f в классе C и все общедоступные методы с именем f в суперклассах класса C.

   Теперь компилятор знает всех возможных кандидатов при вызове метода.
2. Затем компилятор определяет типы параметров, указанных при вызове метода. Если среди всех методов с именем f есть только один метод, типы параметров которого совпадают с указанными, происходит его вызов. Этот процесс называется *разрешением перегрузки* (overloading resolution). Например, при вызове x.f("Hello") компилятор выберет метод f(String), а не метод f(int). Ситуация может осложниться вследствие преобразования типов (int — в double, Manager — в Employee и т.д.). Если компилятор не находит ни одного метода с подходящим набором параметров или в результате преобразования

типов возникает несколько методов, соответствующих данному вызову, выдается сообщение об ошибке.

Теперь компилятор знает имя и типы параметров метода, подлежащего вызову.

Напомним, что имя метода и список типов его параметров образуют *сигнатуру* (signature) метода. Например, методы `f(int)` и `f(String)` имеют одинаковые имена, но разные сигнатуры. Если в подклассе определен метод, сигнатура которого совпадает с сигнатурой некоторого метода суперкласса, то метод подкласса замещает его собой.

Однако возвращаемое значение в сигнатуре не учитывается. В версиях, предшествующих JDK 5.0, тип возвращаемого значения переопределяемого и переопределяющего методов должны были совпадать. Однако теперь допускается, чтобы в подклассе в качестве возвращаемого значения использовался подтип первоначально выбранного типа. Предположим, например, что в классе `Employee` определен метод `getBuddy()`.

```
public Employee getBuddy() { ... }
```

В подклассе `Manager` этот метод переопределен следующим образом:

```
public Manager getBuddy() { ... } // Корректно для JDK 5.0
```

При этом считается, что для методов `getBuddy()` определены *ковариантные* возвращаемые типы.

3. Если метод является закрытым (`private`), статическим (`static`), терминальным (`final`) или конструктором, компилятор точно знает, как его вызвать. (Модификатор `final` описывается в следующем разделе.) Такой процесс называется *статическим связыванием* (static binding). В противном случае метод, подлежащий вызову, определяется по фактическому типу неявного параметра и во время выполнения программы используется *динамическое связывание* (dynamic binding). В нашем примере компилятор сгенерировал бы вызов метода `f(String)` с помощью динамического связывания.
4. Если при выполнении программы для вызова метода используется динамическое связывание, виртуальная машина должна вызвать версию метода, соответствующую *фактическому типу* объекта, на который ссылается переменная `x`. Предположим, что объект имеет фактический тип `D`, являющийся суперклассом класса `C`. Если в классе `D` определен метод `f(String)`, то вызывается именно он. Если нет, то поиск метода `f(String)`, подлежащего вызову, выполняется в суперклассе и т.д.

   На поиск вызываемого метода уходит слишком много времени, поэтому виртуальная машина заранее создает *таблицу методов* (method table) для каждого класса, в которой перечисляются сигнатуры всех методов и фактические методы, подлежащие вызову. При вызове метода виртуальная машина просто просматривает таблицу методов. В нашем примере виртуальная машина проверяет таблицу методов класса `D` и обнаруживает метод `f(String)`, подлежащий вызову. Такими методами могут быть `D.f(String)` или `X.f(String)`, если `X` — некоторый суперкласс класса `D`.

   В этом сценарии есть одна особенность. Если вызывается метод `super.f(args)`, то компилятор просматривает таблицу методов суперкласса неявного параметра.

Рассмотрим подробно вызов метода `e.getSalary()` в листинге 5.1. Переменная `e` имеет тип `Employee`. В этом классе есть только один метод `getSalary()`, у которого нет параметров. Следовательно, в данном случае можно не беспокоиться о разрешении перегрузки.

Поскольку при объявлении метода `getSalary()` не использовались ключевые слова `private`, `static` или `final`, он связывается динамически. Виртуальная машина создает таблицу методов классов `Employee` и `Manager`. Таблица класса `Employee` показывает, что все методы определены в самом классе.

```
Employee:
   getName() -> Employee.getName()
   getSalary() -> Employee.getSalary()
   getHireDday() -> Employee.getHireDay()
   raiseSalary(double) -> Employee.raiseSalary(double)
```

На самом деле это не совсем так. Как мы увидим позднее, класс `Employee` имеет суперкласс `Object`, от которого он наследует большое количество методов; их мы пока будем игнорировать.

Таблица методов класса `Manager` имеет несколько иной вид. В этом классе три метода наследуются, один метод — переопределяется и один — добавляется.

```
Manager:
   getName() -> Employee.getName()
   getSalary() -> Manager.getSalary()
   getHireDday() -> Employee.getHireDay()
   raiseSalary(double) -> Employee.raiseSalary(double)
   setBonus(double) -> Manager.setBonus(double)
```

Во время выполнения программы вызов метода `e.getSalary()` осуществлется следующим образом.

1. Сначала виртуальная машина загружает таблицу, соответствующую реальному типу переменной `e`. Это может быть таблица для `Employee`, `Manager` или другого подкласса класса `Employee`.
2. Затем виртуальная машина определяет класс, в котором определен метод `getSalary()` с соответствующей сигнатурой. Теперь метод, подлежащий вызову, известен.
3. Виртуальная машина вызывает этот метод.

Динамическое связывание обладает одной важной особенностью: оно позволяет модифицировать программы без перекомпиляции их кодов. Это делает программы *динамически расширяемыми* (extensible). Допустим, в программу добавлен новый класс `Executive`, и переменная `e` может ссылаться на объект этого класса. Код, содержащий вызов метода `e.getSalary()`, заново компилировать не нужно. Если *переменная `e` ссылается на объект типа `Executive`, автоматически вызывается метод* `Executive.getSalary()`.

При переопределении метода область видимости метода подкласса должна быть не меньше области видимости переопределяемого метода суперкласса. В частности, если при объявлении метода суперкласса использовалось ключевое слово `public`, метод подкласса также должен быть объявлен как `public`. Программисты часто

ошибаются, забывая указать спецификатор `public` при описании метода подкласса. В этих случаях компилятор сообщает о том, что привилегии доступа к данным ограничены.

Предотвращение наследования: терминальные классы и методы

Иногда наследование является нежелательным. Классы, которые нельзя расширить, называются *терминальными* (`final`). Для того чтобы отметить этот факт, в определении класса используется ключевое слово `final`. Предположим, например, что мы хотим предотвратить создание подклассов класса `Executive`. В этом случае класс определяется следующим образом:

```
final class Executive extends Manager
{
   ...
}
```

Отдельный метод класса также может быть терминальным. Такой метод не может быть переопределен в подклассах. (Все методы терминального класса автоматически являются терминальными.) Пример объявления терминального метода приведен ниже.

```
class Employee
{
   ...
   public final String getName()
   {
     return name;
   }
   ...
}
```

Напомним, что с помощью модификатора `final` могут также описываться поля, являющиеся константами. После создания объекта значение такого поля нельзя изменить. Однако, если класс объявлен как `final`, терминальными становятся только методы; поля в константы не превращаются.

Существует единственный аргумент в пользу применения ключевого слова `final` при объявлении метода или класса: это позволяет гарантировать неизменность семантики. Так, например, в классе `Calendar` методы `getTime()` и `setTime()` являются терминальными. Поступая таким образом, разработчики берут на себя ответственность за корректность преобразования содержимого объекта `Date` в состояние календаря. В подклассах невозможно изменить принцип преобразования. В качестве другого примера можно привести класс `String`. Создавать подклассы этого класса запрещено. Поэтому если у вас есть переменная типа `String`, можете быть уверены, что она ссылается именно на сроку и ни на какой другой объект.

Некоторые программисты считают, что ключевое слово `final` надо применять при объявлении всех методов. Исключением являются только те случаи, когда есть веские основания для использования полиморфизма. Действительно, в методах C++ и C# для применения полиморфизма надо принять специальные меры. Может быть, указанное правило слишком жесткое, но, несомненно, что при формировании иерархии классов надо серьезно заботиться об использовании терминальных методов.

На заре развития Java некоторые программисты пытались использовать ключевое слово `final` для того, чтобы исключить накладные расходы, вызываемые динамическим связыванием. Если метод не переопределяется и размеры его невелики, компилятор применяет процедуру оптимизации, которая состоит в непосредственном включении кода. Например, вызов метода `e.getName()` заменяется обращением к полю `e.name`. Такое преобразование достаточно эффективно, так как ветвление программы несовместимо с упреждающей загрузкой команд, применяемой в процессорах. Следует заметить, что если метод `getName()` будет переопределен, компилятор не сможет непосредственно включить его код, поскольку ему станет неизвестно, какой из методов должен быть вызван в процессе выполнения программы.

Компонент виртуальной машины, называемый *динамическим компилятором* (just-in-time compiler), может выполнять оптимизацию более эффективно, чем обыкновенный компилятор. Ему в точности известно, какие классы расширяют данный класс, и он может проверить, действительно ли метод переопределяется. Если размеры метода невелики, он часто вызывается и не переопределен, динамический компилятор выполнит непосредственное включение кода. Если же виртуальная машина загрузит подкласс, в котором метод переопределяется, непосредственное включение будет отменено. В таких ситуациях выполнение программы существенно замедляется, но возникают они редко.

В языке C++ методы по умолчанию динамически не связываются, поэтому их можно объявлять подставляемыми (inline) и заменять их вызовы соответствующими кодами. Однако механизма, позволяющего предотвратить замещение метода суперкласса методом подкласса, в языке C++ нет. В программе на этом языке можно создать класс, из которого нельзя породить дочерние классы, но для этого нужно выполнить ряд сложных действий, да и кроме того, аргументы в пользу формирования такого класса сомнительны. (При желании вы можете реализовать его в качестве упражнения. Подсказка: воспользуйтесь виртуальным базовым классом.)

Приведение типов

В главе 3 мы рассмотрели процесс принудительного преобразования одного типа в другой, называемый *приведением типов* (casting). Для этого в языке Java существует специальное обозначение. Например, при выполнении следующего фрагмента кода значение переменной `x` преобразуется в целое число путем отбрасывания дробной части.

```
double x = 3.405;
int nx = (int)x;
```

Преобразование числа с плавающей десятичной точкой в целое – достаточно простая операция. В языке программирования должна быть предусмотрена возможность преобразовывать ссылку на объект одного класса в ссылку на объект другого класса. Чтобы выполнить приведение объектной ссылки, используются такие же синтаксические конструкции, как и для числовых выражений. Имя требуемого класса надо поместить в скобки и поставить перед объектной ссылкой, которую нужно преобразовать. Пример соответствующего выражения приведен ниже.

```
Manager boss = (Manager)staff[0];
```

Для приведения типов существует только одна причина – необходимость использовать все возможности объекта после того, как его фактический тип был на время забыт. Например, в классе `ManagerTest` массив `staff` представляет собой массив объектов `Employee`. Этот тип выбран, поскольку *некоторые* элементы массива хранят информацию об обычных сотрудниках. Чтобы получить доступ ко всем новым полям класса `Manager`, нам может понадобиться привести тип некоторых элементов массива к типу `Manager`. (В примере, рассмотренном выше, мы предприняли специальные меры, чтобы избежать приведения типов. Мы инициализировали переменную `boss` объектом `Manager`, перед тем как поместить ее в массив. Для того чтобы задать размер премии, нужно знать правильный тип объекта.)

Как известно, в языке Java тип объектной переменной определяет разновидность объектов, на которые может ссылаться эта переменная. Например, переменная `staff[1]` ссылается на объект `Employee` (поэтому она может ссылаться и на объекты `Manager`).

В процессе работы компилятор проверяет, что происходит в процессе присваивания: сужаются или расширяются возможности объекта, на который указывает переменная. Если переменной суперкласса присваивается объект подкласса, возможности *класса сужаются, и компилятор без проблем позволяет программисту сделать это.* Если, наоборот, объект суперкласса присваивается переменной подкласса, возможности класса расширяются, поэтому программист должен подтвердить это с помощью обозначения, предназначенного для приведения типов, указав в скобках имя подкласса. Таким образом, виртуальная машина получает возможность контролировать действия программиста в процессе выполнения программы.

Что произойдет, если попытаться осуществить приведение типов вниз по цепочке наследования и попробовать обмануть компилятор?

```
Manager boss = (Manager)staff[1]; // ОШИБКА
```

При выполнении программы система обнаружит несоответствие и сгенерирует исключение. Если ее не обработать, работа программы будет прекращена. Итак, перед приведением типов следует проверить его корректность. Для этого нужно использовать оператор `instanceof`. Например:

```
if (staff[i] instanceof Manager)
{
   boss = (Manager)staff[1];
   ...
}
```

В заключение заметим, что компилятор не позволит выполнить некорректное приведение типов. Например, наличие в тексте программы представленной ниже строки приведет к ошибке на этапе компиляции, поскольку класс `Date` не является подклассом класса `Employee`.

```
Date c = (Date)staff[1];
```

Таким образом, можно сформулировать основные правила приведения типов.

- Приведение типов можно выполнять только в иерархии наследования.
- Чтобы проверить корректность приведения суперкласса к подклассу, используйте оператор `instanceof`.

Если в приведенном ниже выражении x содержит значение null, исключение не будет сгенерировано.

```
x instanceof C
```

Выражение лишь вернет значение false. Это вполне разумно. Поскольку нулевая ссылка не относится ни к одному объекту, она, очевидно, не относится ни к одному объекту C.

Часто приведение типа объекта — не самое лучшее решение. В нашем примере выполнять преобразование объекта Employee в объект Manager не обязательно. Метод getSalary() прекрасно работает с обоими типами, поскольку динамическое связывание автоматически определяет правильный метод.

Приведение типов необходимо лишь тогда, когда надо использовать уникальный метод, который есть только в классе Manager, например setBonus(). Если вы, работая с объектом Employee, вдруг захотите вызвать метод setBonus(), задайте себе вопрос: не свидетельствует ли это о недостатках суперкласса? Возможно, следует пересмотреть структуру суперкласса и добавить в него метод setBonus(). Помните, чтобы программа завершила свою работу, достаточно одного-единственного необрабатываемого исключения ClassCastException. В общем, следует по возможности избегать приведения типов и применения оператора instanceof.

В языке Java для приведения типов используется устаревший синтаксис языка C, однако работает он как безопасная операция динамического приведения типов dynamic_cast языка C++. Например, приведенные ниже две строки кода, написанные на разных языках, почти эквивалентны.

```
Manager boss = (Manager)staff[1]; //Язык Java
Manager* boss = dynamic_cast<Manager*>(staff[1]); // Язык C++
```

Между ними существует одно различие. Если приведение типов выполняется неверно, оно не порождает нулевой объект, а генерирует исключение. В этом смысле приведение типов в языке Java напоминает приведение *ссылочных типов* (references) в языке C++. Это очень значительный недостаток. В языке C++ и контроль, и преобразование типов можно выполнить в одном операторе

```
Manager* boss = dynamic_cast<Manager*>(staff[1]); // Язык C++
if (boss != NULL) ...
```

В языке Java используется комбинация оператора instanceof и оператора приведения типов.

```
if (staff[1] instanceof Manager) //Язык Java
{
   Manager boss = (manager)staff[1];
   ...
}
```

Абстрактные классы

Поднимаясь по иерархии наследования, классы становятся более универсальными и более абстрактными. В некотором смысле родительские классы, находящиеся в верхней части иерархии, становятся настолько абстрактными, что их рассматривают как основу

для разработки других классов, а не как класс, позволяющий создавать конкретные объекты. Расширим, например, иерархию, в которую входит класс Employee. Сотрудник — это человек, а человек может быть студентом. Расширим нашу иерархию классов, добавив в нее классы Person и Student.

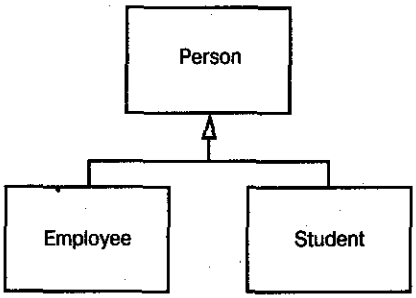


Рис. 5.2. Диаграмма наследования для класса Person и его подклассов

Какие проблемы возникают при таком высоком уровне абстракции? Существуют определенные атрибуты, характерные для каждого человека, например имя. И студенты, и сотрудники имеют имена, и при со+здании общего суперкласса понадобится перенести метод getName() на более высокий уровень в иерархии наследования.

Добавим теперь новый метод getDescription(), предназначенный для создания краткой характеристики человека, например:

```
an employee with a salary of $50,000.00
a student majoring in computer science
```

Для классов Employee и Student этот метод реализуется довольно просто. Однако какую информацию можно поместить в класс Person? В нем ничего нет, кроме имени. Разумеется, можно было бы реализовать метод Person.getDescription(), возвращающий пустую строку. Однако есть способ получше. Используя ключевое слово abstract, можно вовсе не реализовывать этот метод.

```
public abstract String getDescription();
   // Реализация не требуется.
```

Для большей ясности класс, содержащий один или несколько абстрактных методов, можно объявить абстрактным.

```
abstract class Person
{
   ...
   public abstract String getDescription();
}
```

Кроме абстрактных методов, абстрактные классы могут содержать конкретные данные и методы. Например, класс Person хранит имя человека и содержит конкретный метод, возвращающий это имя.

```
abstract class Person
{
   public Person(String n)
```

```
   {
      name = n;
   }

   public abstract String getDescription();

   public String getName()
   {
      return name;
   }

   private String name;
}
```

Многие программисты полагают, что абстрактные классы должны содержать только абстрактные методы. Это далеко не так. В суперклассе следует реализовывать как можно больше функциональных возможностей, независимо от того, абстрактный он или нет. В частности, поля и *неабстрактные* методы, общие для всех подклассов, нужно помещать в абстрактный суперкласс.

Абстрактные методы представляют собой прототипы методов, реализованных в подклассах. Расширяя абстрактный класс, можно оставить некоторые или все абстрактные методы неопределенными. При этом подкласс станет абстрактным. Кроме того, можно определить все методы. Тогда подкласс не будет абстрактным.

Определим класс `Student`, *расширяющий абстрактный класс* `Person` *и реализующий метод* `getDescription()`. Поскольку ни один из методов в классе `Student` не является абстрактным, нет никакой необходимости объявлять сам класс абстрактным.

Класс может быть объявлен абстрактным, даже если он не содержит ни одного абстрактного метода.

Создать объекты абстрактного класса невозможно. Например, приведенное ниже выражение ошибочно.

```
new Person("Vince Vu");
```

Однако можно создать объекты конкретного подкласса.

Заметим, что можно создавать *объектные переменные* абстрактных классов, однако такие переменные должны ссылаться на объект неабстрактного класса. Рассмотрим следующую строку кода:

```
Person p = new Student("Vince Vu", "Economics");
```

где `p` — переменная абстрактного типа `Person`, ссылающаяся на экземпляр неабстрактного подкласса `Student`.

В языке C++ абстрактный метод называется *чисто виртуальной функцией* (pure virtual function). Его обозначение оканчивается символами `=0`.

```
class Person // Язык C++
{
   public:
      virtual string getDescription() = 0;
      ...
};
```

Класс в языке C++ является абстрактным, если он содержит хотя бы одну чисто виртуальную функцию. В языке C++ нет специального ключевого слова, означающего абстрактные классы.

Определим конкретный подкласс Student, расширяющий абстрактный класс Person.

```
class Student extends Person
{
   public Student(String n, String m)
   {
      super(n);
      major = m;
   }

   public String getDescription()
   {
      return "a student majoring in " + major;
   }

   private String major;
}
```

В этом подклассе определяется метод getDescription(). Следовательно, все методы в классе Student являются конкретными, и класс больше не является абстрактным.

В программе, представленной в листинге 5.2, определен абстрактный суперкласс Person и два конкретных подкласса Employee и Student. Заполним массив типа Person ссылками на экземпляры классов Employee и Student.

```
Person[] people = new Person[2];
people[0] = new Employee(...);
people[1] = new Student(...);
```

Затем выведем имена и характеристики этих объектов.

```
for (Person p : people){
   System.out.println(p.getName() + ", " + p.getDescription());
```

Некоторых читателей присутствие вызова p.getDescription()может озадачить. Не относится ли он к неопределенному методу? Учтите, что переменная p никогда не ссылается ни на один объект абстрактного класса Person, поскольку создать такой объект попросту невозможно. Переменная p всегда ссылается на объект конкретного подкласса, например Employee или Student. Для этих объектов метод getDescriprion() определен.

Можно ли пропустить в классе Person все абстрактные методы и определить метод getDescription() в подклассах Employee и Student? Это не будет ошибкой, но тогда метод getDescription() нельзя будет вызвать с помощью переменной p. Компилятор гарантирует, что вызываются только методы, определенные в классе.

Абстрактные методы представляют собой важное понятие языка Java. В основном они применяются при создании интерфейсов, которые будут детально рассмотрены в главе 6.

Листинг 5.2. Содержимое файла `PersonTest.java`

```
import java.text.*;
import java.util.*;


public class PersonTest
{
   public static void main(String[] args)
   {
      Person[] people = new Person[2];

      // Заполняем массив объектами Student и Employee.

      people[0] =
         new Employee("Harry Hacker", 50000, 1989, 10, 1);
      people[1] =
         new Student("Maria Morris", "computer science");

      // Вывод имен и описаний всех объектов Person
      for (Person p : people)
         System.out.println(p.getName() + ", "
            + p.getDescription());
   }
}

abstract class Person
{
   public Person(String n)
   {
      name = n;
   }

   public abstract String getDescription();

   public String getName()
   {
      return name;
   }

   private String name;
}

class Employee extends Person
{
   public Employee(String n, double s,
      int year, int month, int day)
   {
      // Имя передается конструктору суперкласса.
      super(n);
      salary = s;
      GregorianCalendar calendar =
         new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public double getSalary()
```

```
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

   public String getDescription()
   {
      return String.format
        ("an employee with a salary of $%.2f", salary);
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   private double salary;
   private Date hireDay;
}

class Student extends Person
{
   /**
      @param n имя студента
      @param m специализация студента
   */
   public Student(String n, String m)
   {
      // Строка n передается конструктору суперкласса
      super(n);
      major = m;
   }

   public String getDescription()
   {
      return "a student majoring in " + major;
   }

   private String major;
}
```

Защищенный доступ

Как известно, поля в классе желательно объявлять как `private`, а некоторые методы — как `public`. Любые закрытые (`private`) элементы невидимы из других классов. Это утверждение справедливо и для подклассов: подкласс не имеет доступа к закрытым полям суперкласса.

Иногда нужно ограничивать доступ к некоторому методу и открывать его лишь для подклассов. Гораздо реже возникает необходимость предоставлять методам подкласса доступ к полям суперкласса. В этом случае элемент класса объявляется защищенным

с помощью ключевого слова protected. Например, если в суперклассе Employee поле hireDay объявлено защищенным, а не закрытым, методы подкласса Manager смогут обращаться к нему непосредственно.

Однако методы класса Manager имеют доступ лишь к полям hireDay, принадлежащим объектам самого класса Manager, но не класса Employee. Это ограничение введено для того, чтобы программисты не могли злоупотреблять механизмом защищенного доступа и создавать подклассы лишь для получения доступа к защищенным полям.

На практике пользоваться атрибутом protected нужно очень осторожно. Предположим, что ваш класс, в котором есть защищенные поля, используется другими разработчиками. Без вашего ведома другие программисты могут создавать подклассы вашего класса и тем самым получать доступ к защищенным полям. Теперь вы уже не сможете изменять реализацию своего класса, не уведомив других программистов. Это противоречит принципам объектно-ориентированного программирования, поощряющего инкапсуляцию данных.

Применять защищенные методы полезнее, чем защищенные поля. Метод класса можно объявить защищенным, чтобы ограничить его использование. Это значит, что методы подклассов (предки которых известны по определению) могут вызывать этот метод, а методы других классов — нет.

В качестве примера защищенного метода можно привести clone() класса Object (подробно этот вопрос будет рассмотрен в главе 6).

В языке Java защищенные элементы доступны из всех подклассов, а также из других классов того же пакета. Этим Java отличается от языка C++, в котором ключевое слово protected имеет несколько иной смысл. Таким образом, в Java ограничения на доступ к защищенным элементам шире, чем в языке C++.

Итак, в языке Java есть четыре модификатора доступа, управляющих областью видимости.

1. Область видимости ограничена классом (private).
2. Область видимости не ограничена (public).
3. Область видимости ограничена пакетом и всеми подклассами (protected).
4. Область видимости ограничена пакетом (к сожалению, по умолчанию). Никакого модификатора указывать не нужно.

Object: глобальный суперкласс

Класс Object является предком всех классов — каждый класс в языке Java расширяет класс Object. Однако явно отражать этот факт, как в приведенной ниже строке кода, не нужно.

```
class Employee extends Oblects
```

Если суперкласс явно не указан, им считается класс Object. Поскольку в языке Java *каждый* класс расширяет Object, очень важно знать, какими возможностями обладает сам класс Object. Некоторые аспекты класса Object будут изучены в других главах, а остальную информацию читатели могут найти в документации. (Некоторые методы класса Object можно понять лишь после изучения потоков — см. том 2.)

Переменную типа Object можно использовать в качестве ссылки на объект любого типа:

```
Object obj = new Employee("Harry Hacker", 35000);
```

Разумеется, переменная этого класса полезна лишь как средство для хранения значений произвольного типа. Чтобы сделать с этим значением что-то конкретное, нужно знать его исходный тип, а затем выполнить приведение типов:

```
Employee e = (Employee) obj;
```

В языке Java объектами не являются лишь *простые типы*: числа, символы и логические значения.

Все массивы, независимо от того, содержатся ли в его элементах объекты или простые типы, являются объектами, расширяющими класс Object.

```
Employee[] staff = new Employee[10];
obj = staff; // OK
obj = new int[10]; // OK
```

В языке C++ аналогичного глобального базового класса нет, хотя, разумеется, любой указатель можно преобразовать в указатель типа void*.

Метод equals()

Метод equals() класса Object проверяет, эквивалентны ли два объекта. Поскольку метод equals() реализован в классе Object, он определяет лишь, ссылаются ли переменные на один и тот же объект. В качестве проверки по умолчанию эти действия вполне оправданы: всякий объект эквивалентен самому себе. Для некоторых классов большего и не требуется. Например, вряд ли кому-то потребуется анализировать два объекта PrintStream и выяснять, отличаются ли они друг от друга. Однако в ряде случаев эквивалентными должны считаться объекты одного типа, имеющие одинаковые состояния.

Рассмотрим в качестве примера объекты, описывающие сотрудников. Очевидно, что они одинаковы, если совпадают имена, размеры заработной платы и даты приема на работу. (Строго говоря, в реальных системах учета информации о сотрудниках более оправдано сравнение идентификационных номеров. Здесь же мы лишь демонстрируем принципы реализации метода equals().)

```
class Employee
{ // ...
  public boolean equals(Object otherObject)
  {
    // Быстрая проверка идентичности объектов.
    if (this == otherObject) return true;

    // Если явный параметр — null, возвращается значение false.
    if (otherObject == null) return false;

    // Если классы не совпадают, они не эквивалентны.
    if (getClass() != otherObject.getClass())
        return false;
```

```
    // Теперь мы знаем, что объект otherObject
    // имеет тип Employee и не является нулевым.
    Employee other = (Employee) otherObject;

    // Проверим, хранятся ли в полях объектов идентичные значения.
    return name.equals(other.name)
       && salary = other.salary
       && hireDay.equals(other.hireDay);
   }
}
```

Метод `getClass()` определяет тип объекта. Этот метод будет подробно обсуждаться далее в этой главе. Чтобы объекты были эквивалентны, они как минимум должны быть объектами одного и того же класса.

Определяя метод `equals()` для подкласса, надо сначала вызывать тот же метод суперкласса. Если проверка даст отрицательный результат, объекты не могут быть идентичными. Если же поля суперкласса совпадают, можно приступать к сравнению полей подкласса.

```
class Manager extends Employee
{
   . . .
   public boolean equals(Object otherObject)
   {
      if (!super.equals(otherObject)) return false;
      // При выполнении super.equals() проверяется,
      // принадлежит ли otherObject тому же классу
      Manager other = (Manager) otherObject;
      return bonus == other.bonus;
   }
}
```

Проверка эквивалентности объектов и наследование

Как должен работать метод `equals()`, если неявные и явные параметры не принадлежат одному и тому же классу? Мы рекомендуем составлять код так, чтобы метод `equals()` возвращал значение `false`, если классы не совпадают. Однако многие программисты используют следующую проверку:

```
if (!(otherObject instanceof Employee)) return false;
```

При этом остается возможность, что `otherObject` принадлежит подклассу. Данный подход может привести к возникновению проблем. Спецификация языка Java требует, чтобы метод `equals()` обладал следующими характеристиками.

1. *Рефлексивность.* Для любой ненулевой ссылки `x` вызов `x.equals(x)` должен возвращать значение `true`.
2. *Симметричность.* Для любых ссылок `x` и `y` вызов `x.equals(y)` должен возвращать значение `true` тогда и только тогда, когда `y.equals(x)` возвращает `true`.
3. *Транзитивность.* Для любых ссылок `x`, `y` и `z`, если вызовы `x.equals(y)` и `y.equals(z)` возвращают значение `true`, вызов `x.equals(z)` возвращает значение `true`.

4. *Непротиворечивость.* Если объекты, на которые ссылались переменные `x` и `y` не изменяются, то повторный вызов `x.equals(y)` должен возвращать то же значение.
5. Для любой ненулевой ссылки `x` вызов `x.equals(null)` должен возвращать значение `false`.

Целесообразность введения данных правил очевидна. Так, например, очевидно, что результаты проверки не должны зависеть от того, вызывается ли в программе `x.equals(y)` или `y.equals(x)`.

Однако требование симметрии имеет свои особенности в случае, если явный и неявный параметры принадлежат разным классам. Рассмотрим следующий вызов:

```
e.equals(m)
```

Здесь объект `e` принадлежит классу `Employee`, а объект `m` — классу `Manager`, причем каждый из них содержит одинаковые имена, зарплату и дату приема на работу. Если не выполнить проверку эквивалентности классов, которым принадлежат объекты `m` и `e`, метод вернет значение `true`. Однако это значит, что и обратный вызов, `m.equals(e)`, также должен возвращать значение `true`, — правило 2 не позволяет ему возвращать значение `false` или генерировать исключение.

В результате оказывается, что класс `Manager` неоправданно тесно связан с другими классами. Если метод `equals()` используется для сравнения с объектом `Employee`, он не должен принимать во внимание информацию, отличающую менеджера от обычного сотрудника. В данной ситуации оператор `instanceof` выглядит достаточно привлекательно.

Некоторые специалисты считают, что проверка с использованием метода `getClass()` *некорректна, поскольку при этом нарушается принцип подстановки.* В пользу данного мнения приводят метод `equals()` класса `AbstractSet`, который проверяет, содержат ли два множества одинаковые элементы и расположены ли они в одинаковом порядке. Класс `AbstractSet` выступает в роли суперкласса для классов `TreeSet` и `HashSet`, которые не являются абстрактными. Эти классы используют различные алгоритмы для работы с элементами множества. На практике разработчику необходимо иметь возможность сравнивать любые два множества, независимо от того, как они реализованы.

Следует признать, что данный пример слишком специфичен. В данном случае имело бы смысл объявить метод `AbstractSet.equals()` как терминальный, чтобы невозможно было изменить семантику проверки на эквивалентность множеств. (На самом деле при объявлении метода ключевое слово `final` не указано. Разработчики класса оставили возможность для реализации более эффективного алгоритма проверки на эквивалентность.)

Таким образом, вырисовываются два сценария.

- Если проверка эквивалентности реализована в подклассе, правило симметричности требует использования метода `getClass()`.
- Если проверка производится средствами суперкласса, можно применять оператор `instanceof`; при этом становится возможной ситуация, когда два объекта разных классов будут признаны эквивалентными.

В примере с сотрудниками и менеджерами мы считаем два объекта эквивалентными, если их поля совпадают. Если в нашем распоряжении есть два объекта Manager

с одинаковыми именами, заработной платой и датой приема на работу, но с различными суммами премии, мы должны признать объекты разными. Следовательно, мы должны использовать проверку с помощью метода `getClass()`.

Теперь предположим, что для проверки эквивалентности используется идентификационный номер сотрудника. Такая проверка имеет смысл для всех подклассов. В этом случае мы можем применять оператор `instanceof` и объявить метод `Employee.equals()` как `final`.

Ниже приведены рекомендации для создания метода `equals()`.

1. Предположим, что явный параметр называется `otherObject`, – впоследствии его тип нужно будет привести к типу другой переменной, которую назовем `other`.
2. Проверить, идентичны ли ссылки `this` и `otheObject`.

   ```
   if(this == otherObject) return true;
   ```

   Это выражение используется для оптимизации проверки. Гораздо быстрее проверить идентичность ссылок, чем сравнивать поля объектов.
3. Выяснить, является ли ссылка `otherObject` нулевой (`null`). Если да, вернуть значение `false`. Эту проверку нужно делать обязательно.

   ```
   if (otherObject == null) return false;
   ```

   Сравнить классы `this` и `otheObject`. Если семантика проверки может измениться в подклассе, использовать метод `getClass()`.

   ```
   if (getClass() != otherObject.getClass()) return false;
   ```

   Если принцип проверки остается справедливым для всех подклассов, использовать оператор `instanceof`.

   ```
   if (!(otherObject instanceof ClassName)) return false;
   ```

4. Преобразовать объект `otherObject` в переменную требуемого класса.

   ```
   Имя_класса other = (Имя_класса)otherObject;
   ```

5. Теперь нужно сравнить между собой все поля. Для полей простых типов используется оператор `==`, для объектных полей – метод `equals()`. Если все поля двух объектов совпадают друг с другом, возвращается значение `true`, в противном случае – значение `false`.

   ```
   return поле_1 == other. поле_1
   && поле_2.equals(other. поле_2)
   && ...;
   ```

   Если в подклассе вы переопределяете метод `equals()`, в него надо включить вызов `super.equals(other)`.

Стандартная библиотека Java содержит более 150 реализаций метода `equals()`. Некоторые из них включают множество операторов `instanceof`, вызовов метода `getClass()` и фрагментов кода, предназначенных для обработки исключения `ClassCastException`, другие не выполняют практически никаких действий. Создается впечатление, что многие программисты не представляют себе всех особенностей метода `equals()`. Например, класс `Rectangle` представляет собой подкласс

Rectangle2D. В обоих классах реализован метод equals(), причем для проверки используется оператор instanceof. Сравнивая Rectangle2D с Rectangle при условии равенства координат, мы получим значение true, если же поменяем параметры местами, проверка даст значение false.

Реализуя метод equals(), многие программисты допускают стандартную ошибку. Сможете ли вы сказать, какая проблема возникнет при выполнении следующего фрагмента кода?

```
public class Employee
{
   public boolean equals(Employee other)
   {
      return name.equals(other.name)
         && salary == other.salary
         && hireDay.equals(other.hireDay);
   }
   ...
}
```

При определении метода явный параметр объявлен как Employee. В результате он не переопределяет метод equals() класса Object.

Начиная с JDK 5.0, появилась возможность застраховаться от возникновении подобной ошибки, специальным образом указывая, что разрабатываемый метод призван заместить соответствующий метод суперкласса. Для этой цели используется дескриптор @Override.

```
@Override public boolean equals(Object other)
```

Если при этом вы случайно определите новый метод, компилятор вернет сообщение об ошибке. Предположим, что в классе Employee присутствует приведенная ниже строка кода.

```
@Override public boolean equals(Employee other)
```

Поскольку данный метод не переопределяет метод, определенный в суперклассе Object, будет обнаружена ошибка.

Дескриптор @Override относится к языковым конструкциям, применяемым для работы с метаданными. Механизм метаданных достаточно универсальный и допускает расширение. С помощью метаданных компилятор получает возможность отслеживать различные действия. Со временем станет ясно, насколько новые средства облегчат труд разработчиков. Включение в состав компилятора дескриптора @Override — лишь первый шаг в этом направлении.

Метод hashCode()

Хэш-код — это целое число, генерируемое на основе конкретного объекта. Хэш-код можно рассматривать как некоторый шифр: если x и y — разные объекты, то с высокой степенью вероятности должны различаться результаты вызовов x.hashCode() и y.hashCode(). В табл. 5.1 приведено несколько примеров хэш-кодов, полученных в результате вызова метода hashCode() класса String.

Таблица 5.1. Хэш-коды, полученные с помощью метода `hashCode()`

Строка	Хэш-код
Hello	140207504
Harry	140013338
Hacker	884756206

В классе `String` для вычисления хэш-кода используется следующий алгоритм:

```
int hash = 0;
for (int i = 0; i < length(); i++)
   hash = 31 * hash + charAt(i);
```

Метод `hashCode()` определен в классе `Object`. Поэтому каждый объект имеет хэш-код, определяемый по умолчанию. В классе `Object` хэш-код вычисляется на основе адреса памяти, занимаемой объектом. Рассмотрим следующий пример:

```
String s = "Ok";
StringBuffer sb = new StringBuffer(s);
System.out.println(s.hashCode() + " " + sb.hashCode());
String t = new String("Ok");
StringBuffer tb = new StringBuffer(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

Результаты выполнения данного фрагмента кода приведены в табл. 5.2.

Таблица 5.2. Хэш-коды для объектов `String` и `StringBuffer`

Объект	Хэш-код
s	3030
sb	20526976
t	3030
tb	20527144

Заметьте, что строкам s и t соответствуют одинаковые хэш-коды, так как они вычисляются на основе *содержимого* объекта. Для буферов sb и tb хэш-коды различаются. Причина в том, что в классе `StringBuffer` метод `hashCode()` не определен и используется метод `hashCode()` класса `Object`, который определяет хэш-код по адресу занимаемой памяти.

Если вы переопределяете метод `equals()`, вам также следует переопределить и метод `hashCode()`. Он может быть использован при включении объектов в хэш-таблицу. (Хэш-таблицы будут обсуждаться в главе 2 второго тома.)

Метод `hashCode()` возвращает целое число (которое может быть отрицательным). Для того чтобы обеспечить различие хэш-кодов для разных объектов, достаточно объединить хэш-коды полей экземпляра.

Ниже приведен пример метода `hashCode()` для класса `Employee`.

```
class Employee
{
   public int hashCode()
   {
      return 7 * name.hashCode()
```

```
            + 11 * new Double(salary).hashCode()
            + 13 * hireDay.hashCode();
    }
    . . .
}
```

Методы `equals()` и `hashCode()` должны быть совместимы: если `x.equals(y)` возвращает значение `true`, то результаты выполнения `x.hashCode()` и `y.hashCode()` также должны совпадать. Например, если в методе `Employee.equals()` для определения эквивалентности применяется табельный номер сотрудника, то при вычислении хэш-кода также должен использоваться этот номер, а не имя и не адрес памяти, занимаемый объектом.

`java.lang.Object 1.0`

- `int hashCode()`

  Возвращает хэш-код объекта. Хэш-код представляет собой положительное или отрицательное целое число. Эквивалентным объектам должны соответствовать одинаковые хэш-коды.

Метод toString()

Еще одним важным методом класса `Object` является `toString()`, возвращающий значение объекта в виде строки. В качестве примера можно привести метод `toString()` класса `Point`. Он возвращает строку, подобную приведенной ниже.

```
java.awt.Point[x=10,y=20]
```

Большинство (но не все) методы `toString()` возвращают строку, которая состоит из имени класса, за которым указываются значения его полей в квадратных скобках. Ниже приведена реализация метода `toString()` для класса `Employee`.

```
public String toString()
{
   return "Employee[name=" + name
      + ",salary=" + salary
      + ",hireDay=" + hireDay
      + "]";
}
```

На самом деле этот метод можно усовершенствовать. Не будем встраивать имя класса в метод `toString()`, а лишь вызовем метод `getClass().getName()` и получим строку, содержащую имя класса.

```
public String toString()
{
   return getClass().getName()
      + "[name=" + name
      + ",salary=" + salary
      + ",hireDay=" + hireDay
      + "]";
}
```

Теперь метод `toString()` работает и с подклассами.

Разумеется, программист, создающий подкласс, должен определить свой собственный метод `toString()` и добавить поля подкласса. Если в суперклассе используется вызов `getClass().getName()`, подкласс просто вызывает метод `super.ToString()`. Ниже приведен пример метода `toString()` для класса `Manager`.

```
class Manager extends Employee
{
   . . .
   public String toString()
   {
      return super.toString()
        + "[bonus=" + bonus
        + "]";
   }
}
```

Теперь состояние объекта `Manager` выводится следующим образом:

```
Manager[name=...,salary=...,hireDay=...][bonus=...]
```

Метод `toString()` универсален. Есть важная причина для реализации его в каждом классе: если объект объединяется со строкой с помощью оператора "+", компилятор языка Java автоматически вызывает метод `toString()`, чтобы получить представление о его текущем состоянии.

```
Point p = new Point(10, 20);
String message = "The current position is = " + p;
   // Автоматически вызывает метод p.String().
```

Вместо `x.toString()` можно использовать выражение `" " + x`. Конкатенация пустой строки с представлением объекта `x` в виде строки эквивалентна вызову метода `x.toString()`. Такое выражение будет корректным, даже если переменная принадлежит к одному из простых типов.

Предположим, что `x` — произвольный объект и в программе имеется следующая строка:

```
System.out.println(x);
```

В этом случае метод `println()` *вызовет метод* `x.toString()` *и выведет строку результата.*

Метод `toString()`, определенный в классе `Object`, выводит имя класса и адрес объекта. Рассмотрим следующий вызов:

```
System.out.println(System.out);
```

После выполнения метода `println()` отображается следующая строка:

```
java.io.PrintStream@2f6684
```

Как видите, разработчики класса `PrintStream` не позаботились о переопределении метода `toString()`.

Данный метод — отличное средство протоколирования. С его помощью можно получать полезную информацию о состоянии объекта. Так, например, для протоколирования можно применить следующее выражение:

```
System.out.println("Current position = " + position);
```

Прочитав главу 11, вы узнаете, что существует еще лучшее решение.

```
Logger.global.info("Current position = " + position);
```

Мы настоятельно рекомендуем переопределять метод `toString()` в каждом создаваемом вами классе. Это будет полезно как вам, так и программистам, использующим результаты вашей работы.

В листинге 5.3 содержится код методов `equals()` и `toString()`, реализованных для классов `Employee` и `Manager`.

Листинг 5.3. Содержимое файла `EqualsTest.java`

```
import java.util.*;

public class EqualsTest
{
   public static void main(String[] args)
   {
      Employee alice1 = new Employee("Alice Adams",
         75000, 1987, 12, 15);
      Employee alice2 = alice1;
      Employee alice3 = new Employee("Alice Adams",
         75000, 1987, 12, 15);
      Employee bob = new Employee("Bob Brandson",
         50000, 1989, 10, 1);

      System.out.println("alice1 == alice2: " +
         (alice1 == alice2));

      System.out.println("alice1 == alice3: " +
         (alice1 == alice3));

      System.out.println("alice1.equals(alice3): " +
         alice1.equals(alice3));

      System.out.println("alice1.equals(bob): " +
         alice1.equals(bob));

      System.out.println("bob.toString(): " + bob);

      Manager carl = new Manager("Carl Cracker",
         80000, 1987, 12, 15);
      Manager boss = new Manager("Carl Cracker",
         80000, 1987, 12, 15);
      boss.setBonus(5000);
      System.out.println("boss.toString(): " + boss);
      System.out.println("carl.equals(boss): "  +
         carl.equals(boss));
      System.out.println("alice1.hashCode(): " +
         alice1.hashCode());
      System.out.println("alice3.hashCode(): " +
         alice3.hashCode());
      System.out.println("bob.hashCode(): " +
```

```
         bob.hashCode());
      System.out.println("carl.hashCode(): " +
         carl.hashCode());
   }
}

class Employee
{
   public Employee(String n, double s,
      int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar
         = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public boolean equals(Object otherObject)
   {
      // Быстрая проверка идентичности объектов
      if (this == otherObject) return true;

      // Если явный параметр — null, возвращается значение false.
      if (otherObject == null) return false;

      // Если классы не совпадают, они не равны.
      if (getClass() != otherObject.getClass())
         return false;

      // Теперь мы знаем, что otherObject —
      // ненулевой объект Employee.
      Employee other = (Employee)otherObject;

      // Проверка идентичности значений, записанных в полях.
      return name.equals(other.name)
```

```
        && salary == other.salary
        && hireDay.equals(other.hireDay);
  }

  public int hashCode()
  {
    return 7 * name.hashCode()
      + 11 * new Double(salary).hashCode()
      + 13 * hireDay.hashCode();
  }

  public String toString()
  {
     return getClass().getName()
        + "[name=" + name
        + ",salary=" + salary
        + ",hireDay=" + hireDay
        + "]";
  }

  private String name;
  private double salary;
  private Date hireDay;
}

class Manager extends Employee
{
  public Manager(String n, double s,
     int year, int month, int day)
  {
     super(n, s, year, month, day);
     bonus = 0;
  }

  public double getSalary()
  {
     double baseSalary = super.getSalary();
     return baseSalary + bonus;
  }

  public void setBonus(double b)
  {
     bonus = b;
  }

  public boolean equals(Object otherObject)
  {
     if (!super.equals(otherObject)) return false;
     Manager other = (Manager)otherObject;
     // Метод super.equals() проверяет, принадлежат ли объекты
     // this и other одному и тому же классу.
     return bonus == other.bonus;
  }

  public int hashCode()
  {
     return super.hashCode()
```

```
            + 17 * new Double(bonus).hashCode();
   }

   public String toString()
   {
      return super.toString()
         + "[bonus=" + bonus
         + "]";
   }

   private double bonus;
}
```

- `Class getClass()`

  Возвращает класс объекта. Как мы увидим далее в этой главе, в языке Java поддерживается представление классов, инкапсулированное в классе `Class`.

- `boolean equals(Object otherObject)`

  Сравнивает два объекта, возвращает значение `true`, если объекты занимают одну и ту же область памяти, в противном случае возвращается значение `false`. Создавая собственные классы, этот метод следует переопределять.

- `String toString()`

  Возвращает строку, представляющую значение объекта. Создавая собственные классы, этот метод следует переопределять.

- `Object clone()`

  Создает копию (клон) объекта. Исполняющая система Java выделяет память для нового экземпляра и копирует туда содержимое области памяти, в которой хранится текущий объект.

Клонирование объектов имеет большое значение, однако это довольно сложный процесс, изобилующий ловушками, в которые могут попадать неопытные программисты. В главе 6 метод `clone()` будет рассмотрен подробнее.

- `String getName()`

  Возвращает имя класса.

- `Class getSuperclass()`

  Возвращает имя суперкласса данного класса в виде объекта `Class`.

Универсальный класс ArrayList

Во многих языках программирования, в частности в C, размер всех массивов должен задаваться еще на этапе компиляции программы. Это ограничение существенно затрудняет работу программиста. Сколько сотрудников будет работать в отделе? Однозначно — не больше 100 человек. А что если где-то есть отдел со 150 сотрудниками? А если отдел

очень мал и в нем работают только 10 человек? Ведь в этом случае 90% массива не используется!

В языке Java ситуация намного лучше — размер массива можно задавать уже во время выполнения программы.

```
int actualSize = ...;
Employee[] staff = new Employee[actualSize];
```

Разумеется, этот код не решает проблему динамической модификации массивов во время выполнения программы. Задав размер массива, его потом нелегко изменить. Проще всего разрешить эту ситуацию, используя списочные массивы. Для их создания применяются экземпляры класса, который называется `ArrayList`. Данный класс ведет себя как массив, но может динамически изменять размеры по мере добавления новых элементов или удаления существующих. При этом программисту не приходится писать дополнительный код.

В JDK 5.0 `ArrayList` был преобразован в универсальный класс с параметром типа. Тип элемента массива помещается в угловые скобки и добавляется к имени класса. главе 13 будет рассказано о создании собственных универсальных классов, однако пользоваться объектами `ArrayList` можно, не применяя специальных технологий.

Ниже представлена строка кода, посредством которого создается списочный массив, предназначенный для хранения объектов `Employee`.

```
ArrayList<Employee> staff = new ArrayList<Employee>();
```

До появления JDK 5.0 универсальные классы отсутствовали. Вместо них применялся единственный класс `ArrayList`, который позволял хранить объекты типа `Object`. Если вы работаете со старыми версиями Java, не добавляйте к имени `ArrayList` *суффикс в угловых скобках. Его поддерживают только версии JDK, начиная с 5.0.* `ArrayList` можно рассматривать как "низкоуровневый тип".

В старых версиях языка Java для создания массивов, размеры которых динамически изменялись, программисты использовали класс `Vector`. Однако класс `ArrayList` эффективнее. С его появлением исчезла необходимость использовать класс `Vector`.

Для добавления новых элементов в списочный массив используется метод `add()`. Ниже показан код, используемый для заполнения такого массива объектами `Employee`.

```
staff.add(new Employee("Harry Hacker", . . .));
staff.add(new Employee("Tony Tester", . . .));
```

Списочный массив управляет внутренним массивом ссылок на объекты. В конце концов элементы массива могут оказаться исчерпаны. Здесь на помощь приходят специальные средства списочного массива: если метод `add()` вызывается при заполненном внутреннем массиве, `ArrayList` автоматически создает массив большего размера и копирует в него все объекты.

Если вы заранее знаете, сколько элементов нужно хранить, то перед заполнением списка массивов вызовите метод `ensureCapacity()`:

```
staff.ensureCapacity(100);
```

Этот метод выделит память для внутреннего массива, состоящего из 100 объектов. Затем можно вызвать метод `add()`, который не будет иметь проблем с перераспределением памяти.

Количество элементов, которые будут храниться в списке массивов, можно передать конструктору класса `ArrayList` в качестве параметра:

```
ArrayList<Employee> staff = new ArrayList<Employee>(100);
```

Выделение памяти для списочного и обычного массивов происходит по-разному. Так, приведенные ниже выражения не эквивалентны.

```
new ArrayList<Employee>(100) // Емкость списочного массива равна 100.
new Employee[100]  // Размер массива равен 100.
```

Между емкостью списочного массива и размером обычного массива есть существенное различие. Выделяя память для массива из 100 элементов, вы резервируете их для дальнейшего использования. В то же время емкость списочного массива — это всего лишь *возможная* величина. В ходе работы она может быть увеличена (за счет выделения дополнительной памяти), но сразу после создания массив не содержит ни одного элемента.

Метод `size()` возвращает фактическое количество элементов в списочном массиве. Например, в результате приведенного ниже вызова возвращает текущее количество элементов в массиве `staff`.

```
staff.size()
```

Эквивалентное выражение для определения размера обычного массива a выглядит так:

```
a.lenght
```

Если вы уверены, что список массивов будет иметь постоянный размер, можете вызвать метод `trimToSize()`. Этот метод устанавливает размер блока памяти так, чтобы он точно соответствовал количеству элементов, подлежащих хранению. Механизм сборки мусора предотвращает неэффективное использование памяти, освобождая ее излишки.

Если вы использовали метод `trimToSize()`, то в результате добавления элементов блок памяти будет перемещен, что потребует дополнительного времени. Поэтому вызывать данный метод надо лишь в том случае, когда вы уверены, что дополнительные элементы в списочный массив включаться не будут.

Класс `ArrayList` ведет себя так же, как и шаблон `vector` в C++. И `ArrayList`, и `vector` являются универсальными типами. Однако в шаблоне `vector` происходит перегрузка оператора `[]`, что упрощает доступ к элементам. Поскольку в языке Java перегрузка операторов не предусмотрена, следует явным образом вызывать методы. Кроме того, шаблон `vector` в C++ передается по значению. Если `a` и `b` представляют собой два вектора, то выражение `a = b` приведет к созданию нового вектора, длина которого равна `b`. Все элементы `b` будут скопированы в `a`. В результате выполнения того же самого выражения в языке Java переменные `a` и `b` будут ссылаться на один и тот же списочный массив.

- `ArrayList<T>()`

  Создает пустой списочный массив.

- `ArrayList<T>(int initialCapacity)`

  Создает пустой списочный массив, имеющий заданную емкость.

  Параметр `initialCapacity` Первоначальная емкость списочного массива

- `boolean add(T obj)`

  Добавляет элемент в конец массива. Всегда возвращает значение `true`.

  Параметр `obj` Добавляемый элемент

- `int size()`

  Возвращает количество элементов, хранящихся в списочном массиве. (Количество элементов отличается от емкости массива и не превосходит ее.)

- `void ensureCapacity(int capacity)`

  Гарантирует, что список массивов имеет емкость, достаточную для хранения заданного количества элементов без изменения внутреннего массива.

  Параметр `capacity` Требуемая емкость списочного массива

- `void trimToSize()`

  Сокращает емкость списочного массива до его текущего размера.

Доступ к элементам списочных массивов

К сожалению, ничто не дается бесплатно. Удобство, предоставляемое автоматическим регулированием размера списочного массива, компенсируется более сложным синтаксисом, который требуется для доступа к его элементам. По этой причине класс `ArrayList` не является частью языка Java; это лишь вспомогательный класс, помещенный в стандартную библиотеку.

Вместо квадратных скобок, используемых для доступа к элементу массива, программисты вынуждены применять методы `get()` и `set()`.

Например, чтобы задать i-й элемент, используется выражение

```
staff.set(i, harry);
```

При работе с обычным массивом a те же действия выполнялись бы следующим образом:

```
a[i] = harry;
```

Как в обычных, так и в списочных массивах индексы отсчитываются от нуля.

Получить элемент списочного массива можно с помощью метода `get()`.

```
Employee e = staff.get(i);
```

Как известно, при работе с обычными массивами указывается имя массива и индекс в квадратных скобках.

```
Employee e = a[i];
```

В JDK 5.0 появилась возможность осуществлять перебор элементов списочных массивов с помощью цикла "for each".

```
for (Employee e : staff)
   // Выполнение действий с переменной e
```

Ранее цикл, предназначенный для обработки всех элементов списочного массива, выглядел следующим образом:

```
for (int i = 0; i < staff.size(); i++)
{
   Employee e = (Employee) staff.get(i);
   // Выполнение действий с переменной e
}
```

До появления JDK 5.0 универсальные классы отсутствовали, и единственным вариантом возвращаемого значения метода `get()` класса `ArrayList` была ссылка `Object`. Очевидно, что в вызывающем методе приходилось выполнять приведение типов.

```
Employee e = (Employee) staff.get(i);
```

При использовании класса `ArrayList` возможны неприятные ситуации, связанные с тем, что его методы `add()` и `set()` допускают передачу любого типа параметра. Приведенный ниже вызов воспринимается компилятором как корректный.

```
staff.set(i, new Date());
```

Проблемы возникнут лишь тогда, когда вы извлечете объект и попытаетесь преобразовать его в тип `Employee`. При использовании `ArrayList<Employee>` ошибка будет выявлена на этапе компиляции.

Иногда удается сочетать гибкость и удобство доступа к элементам, пользуясь следующим приемом. Создайте список массивов и добавьте в него все нужные элементы.

```
ArrayList<X> list = new ArrayList<X>();
while (...)
{
   x = ...;
   list.add(x);
}
```

Затем, используя метод `toArray()`, скопируйте все элементы в массив.

```
X[] a = new X[list.size()];
list.toArray(a);
```

Не вызывайте метод `list.set(i, x)`, если *размер* списочного массива меньше `i`. Например, следующий код содержит ошибку:

```
ArrayList<Employee> list = new ArrayList<Employee>(100); // Емкость
                  // равна 100, размер равен нулю.
list.set(0, x);   // Элемента с номером 0 еще нет.
```

Заполняя массив, используйте метод `add()`, а метод `set()` применяйте только для замены уже существующего элемента.

Элементы можно добавлять не только в конец списочного массива, но и в его середину.

```
int n = staff.size() / 2;
staff.add(n, e);
```

Элемент, имеющий индекс n, и расположенные за ним элементы сдвигаются, чтобы освободить место для нового элемента. Если новый размер списка массивов после вставки элемента превышает его емкость, происходит копирование массива.

Аналогично можно удалить элемент из середины списочного массива.

```
Employee e = (Employee)staff.remove(n);
```

Элементы, расположенные после удаленного элемента, сдвигаются влево, а размер списка массивов уменьшается на единицу.

Вставка и удаление элементов не слишком эффективны. Для массивов, размеры которых невелики, проблема не возникает. Однако, если при работе с большими объемами данных приходится часто вставлять и удалять элементы, попробуйте вместо списочного массива применить связный список. Способы программирования связных списков описаны в томе 2.

В листинге 5.4 показана модифицированная программа `EmployeeTest` из главы 4. Массив `Employee[]` заменен классом `ArrayList`. Обратите внимание на следующие особенности.

- Задавать размер массива не нужно.
- С помощью метода `add()` можно добавлять сколько угодно элементов.

Если бы вместо цикла "for each" применялся обычный цикл `for`, то в листинге также были бы видны еще два отличия, характерные для работы со списочным массивом.

- Вместо `length` для подсчета количества элементов применялся бы метод `size()`.
- Вместо выражения `a[i]` для доступа к элементу массива использовался бы метод `(Employee)a.get(i)`.

Листинг 5.4. Содержимое файла `ArrayListTest.java`

```
import java.util.*;

public class ArrayListTest
{
  public static void main(String[] args)
  {
     //Заполнение staff тремя объектами Employee.
    ArrayList<Employee> staff = new ArrayList<Employee>();

    staff.add(new Employee("Carl Cracker", 75000,
       1987, 12, 15));
    staff.add(new Employee("Harry Hacker", 50000,
       1989, 10, 1));
    staff.add(new Employee("Tony Tester", 40000,
       1990, 3, 15));

    // Увеличение заработной платы на 5%.
    for (Employee e : staff)
       e.raiseSalary(5);
```

```
    // Вывод информации обо всех сотрудниках.
    for (Employee e : staff)
       System.out.println("name=" + e.getName()
          + ",salary=" + e.getSalary()
          + ",hireDay=" + e.getHireDay());
  }
}

class Employee
{
 public Employee(String n, double s,
    int year, int month, int day)
 {
    name = n;
    salary = s;
    GregorianCalendar calendar
       = new GregorianCalendar(year, month - 1, day);
    hireDay = calendar.getTime();
 }

 public String getName()
 {
    return name;
  }

  public double getSalary()
  {
    return salary;
  }

  public Date getHireDay()
  {
    return hireDay;
  }

  public void raiseSalary(double byPercent)
  {
    double raise = salary * byPercent / 100;
    salary += raise;
  }

  private String name;
  private double salary;
  private Date hireDay;
}
```

API **java.util.ArrayList<T> 1.2**

- `void set(int index, T obj)`

  Помещает значение в элемент списка массивов с указанным индексом, заменяя его предыдущее содержимое.

  Параметры `index` Позиция (число между 0 и `size()-1`)

  `obj` Новое значение

- `T get(int index)`

  Извлекает значение с указанным индексом.

Параметр	`index`	Индекс извлекаемого элемента (число между 0 и `size()-1`)

- `void add(int Index, T obj)`

  Сдвигает существующие элементы для вставки нового элемента.

Параметры	`index`	Позиция вставляемого элемента (число между 0 и `size()-1`)
	`obj`	*Новый элемент*

- `T remove(int index)`

  Удаляет указанный элемент и сдвигает элементы, следующие за ним. Метод возвращает удаленный элемент.

Параметр	`index`	Позиция удаляемого элемента (число между 0 и `size()-1`)

Совместимость между типизированными и обычными ссылочными массивами

Если вы собираетесь обрабатывать исходный код вашей программы с помощью JDK 5.0 или более поздних версий данного пакета, то можете при создании списочных массивов указывать тип элементов, которые могут храниться в нем. Однако в ряде случаев приходится обеспечивать совместимость с уже существующим кодом, в котором использованы низкоуровневые списочные массивы, реализуемые с помощью класса `ArrayList`.

Предположим, что в програме присутствует следующий код:

```
public class EmployeeDB
{
   public void update(ArrayList list) { ... }
   public ArrayList find(String query) { ... }
}
```

Вы можете указать в качестве параметра метода `update()` типизированный ссылочный массив.

```
ArrayList<Employee> staff = ...;
employeeDB.update(staff);
```

Несмотря на то что компилятор не считает данное выражение ошибкой и даже не выводит предупреждающее сообщение, такой подход нельзя считать полностью безопасным. Метод `update()` *может добавлять в списочный массив элементы, типы которых отличаются от* `Employee`*. При извлечении этих элементов будет сгенерировано исключение. Причины такого поведения программы могут быть непонятны, но следует учитывать, что часть кода разрабатывалась для версий JDK, предшествующих 5.0. Совместимость виртуальной машины Java должна быть обеспечена в первую очередь, потому что, хотя уровень безопасности и не снижается, воспользоваться преимуществом проверки типов на этапе компиляции не удается.*

Если же вы попытаетесь использовать низкоуровневый списочный массив вместо типизированного, компилятор отобразит предупреждающее сообщение.

```
// При обработке следующей строки будет выведено
// предупреждающее сообщение
ArrayList<Employee> result = employeeDB.find(query);
```

Чтобы увидеть текст предупреждающего сообщения, надо указать при вызове компилятора опцию `-Xlint:unchecked`.

Попытка выполнить приведение типов не исправит ситуацию.

```
// На этот раз появится другое предупреждающее сообщение
ArrayList<Employee> result =
   (ArrayList<Employee>) employeeDB.find(query);
```

Изменится лишь сообщение. На этот раз оно информирует о несогласованном приведении типов.

Данная ситуация возникает из-за некоторых ограничений языка Java. Для совместимости компилятор после проверки соблюдения правил работы с типами преобразует типизированные списочные массивы в обычные объекты `ArrayList`. В процессе выполнения программы все списочные массивы одинаковы: виртуальная машина не получает информации о типах. Поэтому приведения типов `(ArrayList)` и `(ArrayList<Employee>)` при работе программы обрабатываются одинаково.

В таких условиях от ваших действий мало что зависит. Модифицируя имеющийся код, вам остается следить за сообщениями компилятора и успокаивать себя тем, что они не свидетельствуют о наличии серьезных ошибок.

Интерфейсные классы и преобразование простых типов

Иногда приходится преобразовывать переменные простых типов в объекты. Все простые типы имеют аналоги в виде классов. Например, существует класс `Integer`, соответствующий типу `int`. Такие классы принято называть *интерфейсными*, или *классами оболочки* (object wrapper). Они имеют очевидные имена: `Integer`, `Long`, `Float`, `Double`, `Short`, `Byte`, `Character`, `Void` и `Boolean`. (Первые шесть классов представляют собой подклассы класса `Number`.) Интерфейсные классы являются терминальными. Таким образом, вы не можете, например, переопределить метод `toString()` в классе `Integer`, чтобы отобразить число римскими цифрами. Кроме того, изменить значение, хранящееся в объекте интерфейсного класса, также невозможно.

Предположим, мы хотим, чтобы в списочном массиве хранились целые числа. К сожалению, с помощью параметра в угловых скобках невозможно задать простой тип, например, выражение `ArrayList<int>` недопустимо. Здесь приходит на помощь интерфейсный класс. Можно объявить списочный массив, предназначенный для хранения объектов `Integer`.

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

С появлением JDK 5.0 стало проще добавлять элементы в массив и извлекать их. Рассмотрим следующую строку кода:

```
list.add(3);
```

При использовании объекта `ArrayList<Integer>` производительность становится меньше, чем при работе с массивом `int[]`. Причина очевидна: каждое значение инкапсулировано внутри объекта, и для его записи или извлечения необходимо выполнять дополнительные действия. Таким образом, использование интерфейсных классов оправдано в небольших наборах данных, когда удобство работы программиста важнее эффективности работы программы.

Она автоматически преобразуется в выражение

```
list.add(new Integer(3));
```

Подобное автоматическое преобразование выполняется для всех простых типов.

Если вы присвоите объект `Integer` переменной `int`, целочисленное значение будет автоматически извлечено из объекта. Другими словами, компилятор преобразует строку кода

```
int n = list.get(i);
```

в выражение

```
int n = list.get(i).intValue();
```

Автоматическое преобразование простого типа в объект и извлечение значения может выполняться и при вычислении арифметических выражений. Например, вы можете применить оператор инкрементирования к переменной, содержащей ссылку на объект `Integer`.

```
Integer n = 3;
n++;
```

Компилятор автоматически извлечет целочисленное значение из объекта, увеличит его на единицу и снова поместит в объект.

На первый взгляд может показаться, что интерфейсные классы ничем не отличаются от соответствующих им простых типов. Различие становится ясно видно при использовании оператора проверки на равенство. Как вы знаете, оператор ==, будучи примененным к объекту, проверяет, ссылаются ли переменные на один и тот же адрес памяти. Ниже приведен пример, в котором, несмотря на равенство целочисленных значений, проверка, вероятнее всего, даст отрицательный результат.

```
Integer a = 1000;
Integer b = 1000;
if (a == b) ...
```

Почему "вероятнее всего", а не "наверняка"? Дело в том, что в некоторых реализациях Java в качестве часто используемого значения применяется один и тот же объект. В этом случае в ходе проверки будет определено, что объекты идентичны. Такая неоднозначность результатов не может приветствоваться разработчиками. Выход – использовать при сравнении интерфейсных объектов метод `equals()`.

Следует заметить, что за преобразование простых типов в интерфейсные объекты и извлечение значений из объектов отвечает не виртуальная машина, а *компилятор*. Он включает в программу необходимые вызовы, а виртуальная машина лишь выполняет байтовые коды.

Интерфейсные классы были реализованы еще в JDK 1.0, но до появления JDK 5.0 приходилось составлять код для преобразования простых типов в объекты и извлекать значения из объектов вручную.

Создавая интерфейсные классы, разработчики Java стремились не только обеспечить хранение чисел, символов и логических значений. По их мнению, в составе интерфейсных классов удобно было реализовать методы для преобразования строк в числа.

Чтобы преобразовать строку в целое число, используется выражение, подобное приведенному ниже.

```
int x = Integer.parseInt(s);
```

Заметьте, что при этом создавать объект `Integer` не обязательно; `parseInt()` представляет собой статический метод.

Ниже описаны наиболее важные методы класса `Integer`. Аналогичные методы есть и в других классах, предназначенных для хранения числовых значений.

Некоторые программисты считают, что с помощью интерфейсных классов можно реализовать методы, модифицирующие свои числовые параметры. Как было сказано в главе 4, на языке Java невозможно написать метод, увеличивающий целое число, передаваемое ему в качестве параметра, поскольку все параметры передаются по значению.

```
public static void triple(int x)   // Неправильно.
{
   x++;   // Увеличивает локальную копию.
}
```

Но, может быть, это ограничение можно обойти, используя вместо типа `int` класс `Integer`?

```
public static void triple(Integer x)   // Неправильно.
{
   ...
}
```

Проблема в том, что объект `Integer` не позволяет изменять содержащуюся в нем информацию. Таким образом, изменить числовой параметр, передаваемый методу, невозможно.

Если вам все же необходимо создать метод, изменяющий свои числовые параметры, используйте один из вспомогательных типов, определенных в пакете `org.omg.CORBA`. К этим типам относятся `IntHolder`, `BooleanHolder` и др. Каждый такой тип содержит общедоступное (!) поле `value`, через которое можно обращаться к хранящемуся в нем числу.

```
public static void triple(IntHolder x)
{
   x.value++;
}
```

- `int intValue()`

  Возвращает значение объекта `Integer` в виде числа типа `int` (данный метод переопределяет метод `intValue()` класса `Number`).

- `static String toString(int i)`

  Возвращает новый объект `String`, представляющий число в десятичном виде.

- `static String toString(int i, int radix)`

  Возвращает новый объект типа `String`, представляющий число в системе счисления, определяемой параметром `radix`.

- `static int parseInt(String s)`

  Возвращает целое значение. Предполагается, что объект типа `String` содержит строку, представляющую целое десятичное число.

- `static int parseInt(String s, int radix)`

  Возвращает целое значение. Предполагается, что объект `String` содержит строку, представляющую целое число в системе счисления, которая задается параметром `radix`.

- `static Integer valueOf(String s)`

  Возвращает новый объект типа `Integer`, инициализированный целым значением, которое задается с помощью параметра. Предполагается, что объект типа `String` содержит строку, представляющую целое десятичное число.

- `static Integer valueOf(String s, int radix)`

  Возвращает новый объект типа `Integer`, инициализированный целым значением, которое задается с помощью первого параметра. Предполагается, что объект `String` содержит строку, представляющую целое число в системе счисления, *которая задается параметром* `radix`.

`java.text.NumberFormat` 1.1

- `Number parse(String s)`

  Возвращает числовое значение, полученное в результате разбора параметра. Предполагается, что объект типа `String` содержит строку, представляющую целое десятичное число.

Методы с переменным числом параметров

До появления JDK 5.0 число параметров любого Java-метода было фиксировано. Теперь есть возможность создавать методы, позволяющие при разных вызовах задавать различное количество параметров.

С одним из таких методов, `printf()`, вы уже знакомы. Ниже представлены два обращения к нему.

```
System.out.printf("%d", n);
System.out.printf("%d %s", n, "widgets");
```

В обеих строках вызывается один и тот же метод, но в первом случае методу передаются два параметра, а во втором случае — три. Определение метода `printf()` выглядит следующим образом:

```
public class PrintStream
{
   public PrintStream printf(String fmt, Object... args)
```

```
    { return format(fmt, args); }
}
```

Здесь многоточие (...) — это часть Java-кода. Оно указывает на то, что в дополнение к параметру `fmt` можно указывать любое число объектов. (Если вызывающий метод передает целочисленное значение или значение одного из простых типов, оно автоматически преобразуется в интерфейсный объект.) После этого метод решает непростую задачу разбора строки `fmt` и связывания спецификаторов формата со значениями `args[i]`.

В методе `printf()` тип параметра `Object...` означает то же самое, что и `Object[]`.

Компилятор при обработке исходного кода выявляет каждое обращение к `printf()`, помещает параметры в массив и, если необходимо, выполняет преобразование простых типов в интерфейсные объекты.

```
System.out.printf("%d %d",
   new Object[] { new Integer(d), "widgets" } );
```

При необходимости вы можете создавать собственные методы с переменным числом параметров и указывать любые типы параметров, в том числе простые типы. Ниже приведен пример простого метода, который вычисляет и возвращает максимальное из произвольного количества значений.

```
public static double max(double... values)
{
   double largest = Double.MIN_VALUE;
   for (double v : values) if (v > largest) largest = v;
   return largest;
}
```

Вызов функции может выглядеть следующим образом:

```
double m = max(3.1, 40.4, -5);
```

Компилятор передает функции `max()` выражение `new double[] { 3.1, 40.4, -5 }`.

В качестве последнего параметра метода, для которого число параметров может быть переменным, допустимо задавать массив.

```
System.out.printf("%d %s", new Object[] { new Integer(1), "widgets" } );
```

Таким образом, вы можете преобразовать существующую функцию, последний параметр которой является массивом, в метод с переменным числом параметров, не изменяя существующего кода. Подобным образом в JDK 5.0 был расширен метод `MessageFormat.format()`.

Отражение

Библиотека отражения (reflection library) предоставляет богатый набор инструментальных средств для динамической работы с Java-кодом. Эта возможность широко используется в JavaBeans при создании компонентов. (Средства JavaBeans будут рассматриваться во втором томе.) Благодаря отражению появляется возможность поддерживать инструментальные средства, подобные тем, которыми пользуются программисты, работающие на Visual Basic. Так, например, когда в процессе разработки или выполнения программы добавляется новый класс, можно организовать опрос с целью выяснить возможности нового класса.

Программа, которая способна анализировать возможности классов, называется *рефлективной* (reflective). Отражение — очень мощный механизм, который можно использовать для решения следующих задач.

- Анализ возможностей классов в процессе выполнения программы.
- Проверка объектов при выполнении программы, например, с его помощью можно реализовать метод `toString()`, совместимый *со всеми* классами.
- Реализация универсального массива для работы с кодом.
- Применение объектов `Method`, которые работают подобно указателями на функции в языках, подобных C++.

Отражение — не только мощный, но и сложный механизм. Он интересен в основном разработчикам инструментальных средств; программисты, создающие обычные прикладные программы, в большинстве случаев могут не применять его. Если вы занимаетесь разработкой приложений и еще не готовы воспринимать сложные вопросы, то можете пропустить остаток этой главы и вернуться к соответствующим разделам позже.

Класс Class

Во время выполнения программы исполняющая система Java всегда осуществляет проверку типов объектов любых классов. Эта информация используется виртуальной машиной для выбора правильного метода, подлежащего вызову.

Получить доступ к этой информации можно иначе — с помощью специального класса, который носит довольно странное имя — `Class`. Метод `getClass()` класса `Object` возвращает экземпляр типа `Class`.

```
Employee e;
...
Class cl = e.getClass();
```

Аналогично тому, как созданный нами объект типа `Employee` описывает свойства конкретного сотрудника, объект `Class` описывает свойства конкретного класса. Вероятно, наиболее популярным методом класса `Class` является метод `getName()`, возвращающий имя класса. Например, оператор

```
System.out.println(e.getClass().getName() + " " + e.getName());
```

выводит строку

```
Employee Harry Hacker
```

если объект `e` относится к классу `Employee`, или

```
Manager Harry Hacker
```

если объект `e` относится к классу `Manager`.

Используя статический метод `forName()`, можно также получить объект `Class`, соответствующий строке.

```
String className = "java.util.Date";
Class cl = Class.forName(className);
```

Этот метод можно применять, если имя класса хранится в строке, содержимое которой во время выполнения программы изменяется. Он работает правильно, если `className`

является именем класса или интерфейса. В противном случае метод `forName()` генерирует *контролируемое исключение* (checked exception). Правила применения *обработчика исключения* (exception handler) при работе с этим методом описаны ниже.

При запуске программы сначала загружается метод `main()`. Он загружает все необходимые классы. Каждый из классов в свою очередь загружает необходимые ему классы и т.д. Если размеры приложения достаточно велики, этот процесс может занять значительное время, причем пользователь вынужден будет ожидать окончания загрузки. Для того чтобы вызвать у пользователя иллюзию быстрого старта, можно воспользоваться следующим приемом. Убедитесь, что класс, содержащий метод `main()`, не ссылается явно на другие классы. Начинайте выводить на экран данные и загрузите остальные классы с помощью вызовов метода `Class.forName()`.

Третий способ получения объекта `Class` прост и удобен. Если `T` — это некоторый тип, то `T.class` — объект соответствующего класса. Например:

```
Class cl1 = Date.class; // Если импортирован java.util.*;
Class cl2 = int.class;
Class cl3 = Double[].class;
```

Отметим, что объект `Class` фактически описывает *тип*, который не обязательно является классом. Например, тип `int` — это не класс, однако, несмотря на это, `int.class` — это объект `Class`.

Начиная с JDK 5.0, класс `Class` допускает указание типа. Например, `Employee.class` соответствует типу `Class<Employee>`. Мы не будем подробно обсуждать этот вопрос, так как использование типов еще больше усложнит и без того сложные понятия. В процессе работы вы вполне можете игнорировать параметр, определяющий тип, и работать с обычным классом `Class`. Дополнительную информацию о типизованных классах `Class` вы найдете в главе 13.

Исторически сложилось так, что метод `getName()` возвращает для массивов довольно странные имена.

- `Double[].class.getName()` возвращает `"[Ljava.lang.Double;"`.
- `int[].class.getName()` возвращает `"[I"`.

Виртуальная машина Java поддерживает уникальный объект `Class` для каждого типа. Следовательно, для сравнения объектов можно использовать оператор `==`, как показано ниже.

```
if (e.getClass() == Employee.class) . . .
```

Еще один полезный метод позволяет создавать в процессе работы программы новый экземпляр класса. Имя метода, как и следует ожидать, — `newInstance()`. Пример его вызова приведен ниже.

```
e.getClass().newInstance();
```

В результате выполнения представленного выражения создается новый экземпляр того же класса, что и `e`. Для инициализации вновь созданного объекта метод

newInstance() использует конструктор по умолчанию (т.е. конструктор без параметров). Если в классе конструктор по умолчанию отсутствует, генерируется исключение.

Используя методы forName() и newInstance(), можно создавать экземпляры классов, имена которых хранятся в строках.

```
String s = "java.util.Date";
Object m = Class.forName(s).newInstance();
```

Если при создании объекта на основе имени класса нужно передать конструктору какие-либо параметры, использовать приведенные выше выражения нельзя. Вместо этого нужно применять метод newInstance() класса Constructor. (Это один из нескольких классов, определенных в пакете java.lang.reflect. Отражение будет обсуждаться в следующем разделе.)

Метод newInstance() является аналогом *виртуального конструктора* в языке C++. Хотя в языке C++ на самом деле нет виртуальных конструкторов в чистом виде, такое выражение употребляется для обозначения методов, реализуемых с помощью специализированных библиотек. Класс Class соответствует классу type_info в языке C++, а метод getClass() — оператору typeid. Класс Class в языке Java более гибок, чем его аналог в языке C++. Класс type_info может только извлекать строку с именем типа, но не способен создавать объекты этого типа.

Обработка исключений

Обработка исключений детально рассматривается в главе 11, но в предшествующих главах иногда будут встречаться методы, при возникновении которых могут возникать исключения.

Если во время выполнения программы возникает ошибка, программа *генерирует исключение*. Это более гибкий процесс, чем простое прекращение выполнения программы, поскольку программист может создавать *обработчики* (handlers) исключений и программа будет продолжать работу.

Если обработчик не предусмотрен, система завершает программу и выводит на экран сообщение о типе исключения. Возможно, вы уже сталкивались с подобными сообщениями об исключениях, например, если пытались использовать ссылку null, обращались за пределы массива.

Существуют два типа исключений: *неконтролируемые* (uncheked) и *контролируемые* (checked). При возникновении контролируемого исключения компилятор проверяет, предусмотрен ли для него обработчик. Однако большинство исключений являются неконтролируемыми. К ним относится, например, ситуация, возникающая при обращении к ссылке null. Компилятор не проверяет, предусмотрен ли программистом обработчик. В принципе надо стараться избегать подобных ошибок, а не программировать их обработку.

Однако не всех ошибок можно избежать. Если, несмотря на все ваши усилия, остался фрагмент кода, при выполнении которого может быть сгенерировано исключение и вы не предусмотрели его обработку, компилятор будет настаивать на том, чтобы вы создали обработчик. Например, метод Class.forName() может сгенерировать контролируемое исключение. В главе 11 мы рассмотрим несколько стратегий обработки исключений, а пока изучим реализацию простейших обработчиков.

Поместим одно или несколько выражений, при выполнении которых могут возникнуть контролируемые исключения, в блок try, а код обработчика — в блок catch.

```
try
{
 Операторы, которые могут генерировать исключение.
}
catch(Exception e)
{
   Код обработчика.
}
```

Рассмотрим пример:

```
try
{
 String name = ...; // Определение имени класса.
 Class cl = Class.forName(name); // Может генерировать исключение.
  ...          // Действия с классом cl.
}
catch(Exception e)
{
 e.printStackTrace();
}
```

Если имя класса не существует, оставшаяся часть блока, расположенного в блоке `try`, игнорируется и управление передается блоку `catch`. (В данном случае предусмотрен вывод содержимого стека с помощью метода `StackTrace()` класса `Throwable`. Этот класс является суперклассом класса `Exception`.) Если в блоке `try` исключение не генерируется, то код обработчика в разделе `catch` не выполняется.

От программиста требуется лишь предусмотреть обработчик для проверяемого исключения. Метод, ответственный за возникновение исключения, обнаружить легко. Если в программе вызывается метод, который может сгенерировать исключение, а обработчика для него нет, то компилятор выведет соответствующее сообщение.

java.lang.Class 1.0

- `static Class forName(String className)`

  Возвращает объект `Class`, представляющий класс с именем `className`.

- `Object newInstance()`

  Возвращает новый экземпляр класса.

java.lang.reflect.Constructor 1.1

- `Object newInstance(Object[] args)`

  Создает новый экземпляр класса.

  Параметр `args` Параметры, передаваемые конструктору

java.lang.Throwable 1.0

- `void printStackTrace()`

  Выводит в стандартный поток ошибок объект `Throwable` и содержимое стека.

Использование отражения для анализа свойств классов

Ниже приводится краткий обзор важных характеристик механизма отражения, позволяющего программисту анализировать устройство класса.

Три класса, `Field`, `Method` и `Constructor`, в пакете `java.lang.reflect` описывают соответственно поля, методы и конструкторы класса. Все три класса содержат метод `getName()`, возвращающий имя анализируемого класса. В состав класса `Field` входит метод `getType()`, возвращающий объект типа `Class`, описывающий тип поля. Классы `Method` и `Constructor` имеют методы, определяющие типы параметров, а класс `Method` может также определять тип возвращаемого значения. Все три класса содержат метод `getModifiers()`, возвращающий целое число, которое соответствует использованным модификаторам, например `public` или `static`. Для анализа этого числа применяются статические методы класса `Modifiers` из пакета `java.lang.reflect`. Например, в этом классе есть методы `isPublic()`, `isPrivate()` и `isFinal()`, определяющие, является тот или иной метод либо конструктор общедоступным, закрытым или терминальным. Все, что нужно сделать, — это применить соответствующий метод класса `Modifier` к целому числу, которое возвращается методом `getModifiers()`. Для вывода самих модификаторов используется метод `Modifier.toString()`.

Методы `getFields()`, `getMethods()` и `getConstructors()` класса `Class` возвращают массивы *общедоступных* полей, методов и конструкторов, принадлежащих анализируемому классу. К ним относятся и *общедоступные* поля суперклассов. Методы `getDeclaredFields()`, `getDeclaredMethods()` и `getDeclaredConstructors()` класса `Class` возвращают массивы, состоящие из всех полей, методов и конструкторов, объявленных в классе. К ним относятся закрытые и защищенные элементы, но не элементы суперклассов.

В листинге 5.5 показано, как отобразить всю информацию о классе. Программа предлагает пользователю ввести имя класса, а затем выводит сигнатуры всех методов и конструкторов вместе с именами всех полей класса. Предположим, что пользователь задал приведенную ниже строку.

```
java.lang.Double
```

В результате на экране появится следующий текст:

```
class java.lang.Double extends java.lang.Number
{
   public java.lang.Double(java.lang.String);
   public java.lang.Double(double);

   public int hashCode();
   public int compareTo(java.lang.Object);
   public int compareTo(java.lang.Double);
   public boolean equals(java.lang.Object);
   public java.lang.String toString();
   public static java.lang.String toString(double);
   public static java.lang.Double valueOf(java.lang.String);
   public static boolean isNaN(double);
   public boolean isNaN();
   public static boolean isInfinite(double);
   public boolean isInfinite();
   public byte buteValue();
   public short shortValue();
```

```
   public int intValue();
   public long longValue();
   public float floatValue();
   public double doubleValue();
   public static double parseDouble(java.lang.String);
   public static native long doubleToLongBits(double);
   public static native long doubleToRawLongBits(double);
   public static native double LongBitsToDouble(long);

   public static final double POSITIVE_INFINITY;
   public static final double NEGATIVE_INFINITY;
   public static final double NaN;
   public static final double MAX_VALUE;
   public static final double MIN_VALUE;
   public static final java.lang.Class.Type;
   private static final long serialVersionUID;
}
```

Эта программа замечательна тем, что она может анализировать любой класс, загружаемый интерпретатором языка Java, а не только классы, доступные при компиляции. В следующей главе мы применим ее для анализа внутренних классов, автоматически генерируемых компилятором языка Java.

Листинг 5.5. Содержимое файла `ReflectionTest.java`

```
import java.util.*;
import java.lang.reflect.*;

public class ReflectionTest
{
   public static void main(String[] args)
   {
      // Имя класса извлекается из командной строки
      // или вводится с клавиатуры.
      String name;
      if (args.length > 0)
         name = args[0];
      else
      {
         Scanner in = new Scanner(System.in);
         System.out.println(
            "Enter class name (e.g. java.util.Date): ");
         name = in.next();
      }

      try
      {
         // Вывод имени класса и суперкласса (if != Object).
         Class cl = Class.forName(name);
         Class supercl = cl.getSuperclass();
         System.out.print("class " + name);
         if (supercl != null && supercl != Object.class)
            System.out.print(" extends " + supercl.getName());

         System.out.print("\n{\n");
         printConstructors(cl);
```

```
            System.out.println();
            printMethods(cl);
            System.out.println();
            printFields(cl);
            System.out.println("}");
         }
         catch(ClassNotFoundException e) { e.printStackTrace(); }
         System.exit(0);
      }

      /**
         Вывод сигнатур всех конструкторов класса.
         @param cl класс
       */
      public static void printConstructors(Class cl)
      {
         Constructor[] constructors = cl.getDeclaredConstructors();

         for (Constructor c : constructors)
         {
            String name = c.getName();
            System.out.print("    " +
               Modifier.toString(c.getModifiers()));
            System.out.print(" " + name + "(");

            // Вывод типов параметров
            Class[] paramTypes = c.getParameterTypes();
            for (int j = 0; j < paramTypes.length; j++)
            {
               if (j > 0) System.out.print(", ");
               System.out.print(paramTypes[j].getName());
            }
            System.out.println(");");
         }
      }

      /**
         Вывод сигнатур всех методов класса.
         @param cl класс
       */
      public static void printMethods(Class cl)
      {
         Method[] methods = cl.getDeclaredMethods();

         for (Method m : methods)
         {
            Class retType = m.getReturnType();
            String name = m.getName();

            // Вывод модификаторов, типа возвращаемого значения
            // и имени метода
            System.out.print("    " +
               Modifier.toString(m.getModifiers()));
            System.out.print(" " + retType.getName() + " " +
               name + "(");
```

```
            // Вывод типов параметров
            Class[] paramTypes = m.getParameterTypes();
            for (int j = 0; j < paramTypes.length; j++)
            {
               if (j > 0) System.out.print(", ");
               System.out.print(paramTypes[j].getName());
            }
            System.out.println(");");
         }
      }

      /**
         Вывод полей класса
         @param cl класс
       */
      public static void printFields(Class cl)
      {
         Field[] fields = cl.getDeclaredFields();

         for (Field f : fields)
         {
            Class type = f.getType();
            String name = f.getName();
            System.out.print("   " +
               Modifier.toString(f.getModifiers()));
            System.out.println(" " + type.getName() + " " +
               name + ";");
         }
      }
}
```

API **java.lang.Class 1.0**

- `Field[] getFields()` **1.1**
- `Field[] getDeclaredFields()` **1.1**

  Метод `getFields()` возвращает массив, содержащий объекты `Field`, соответствующие общедоступным полям анализируемого класса или его суперкласса. Метод `getDeclaredFields()` возвращает массив, содержащий объекты `Field`, соответствующие всем полям анализируемого класса. Метод возвращает массив нулевой длины, если таких полей нет или если объект `Class` представляет собой простой тип или массив.

- `Method[] getMethods()` **1.1**
- `Method[] gteDeclaredMethods()` **1.1**

  Данные методы возвращают массив, содержащий объекты `Method`, соответствующие либо только общедоступным методам, включая наследуемые (`getMethods()`), либо всем методам, за исключением унаследованных (`getDeclaredMethods()`), анализируемого класса или интерфейса.

- `Constructor[] getConstructors()` **1.1**
- `Constructor[] getDeclaredConstructors()` **1.1**

Каждый из этих методов возвращает массив, содержащий объекты `Constructor`, соответствующие либо только общедоступным (`getConstructors()`), либо всем конструкторам (`getDeclaredMethods()`) класса, представленного объектом типа `Class`.

java.lang.reflect.Field 1.1

java.lang.reflect.Method 1.1

java.lang.reflect.Constructor 1.1

- `Class getDeclaringClass()`

  Возвращает объект `Class`, соответствующий классу, в котором определен заданный конструктор, метод или поле.

- `Class[] getExceptionTypes()` (в классах `Constructor` или `Method`)

  Возвращает массив объектов типа `Class`, представляющих собой типы исключений, генерируемых заданным методом.

- `int getModifiers()`

  Возвращает целое число, соответствующее модификатору заданного конструктора, метода или поля. Для анализа возвращаемого значения надо использовать методы класса `Modifier`.

- `String getName()`

  Возвращает строку, в которой содержится имя конструктора, метода или поля.

- `Class[] getParameterTypes()` (в классах `Constructor` или `Method`)

  Возвращает массив объектов типа `Class`, представляющих типы параметров.

- `Class getReturnType()` (в классе `Method`)

  Возвращает объект `Class`, соответствующий возвращаемому типу.

java.lang.reflect.Modifier 1.1

- `static String toString(int modifiers)`

  Возвращает строку с модификаторами, соответствующими битам в целом числе `modifiers`.

- `static boolean isAbstract (int modifiers)`
- `static boolean isFinal (int modifiers)`
- `static boolean isInterface (int modifiers)`
- `static boolean isNative (int modifiers)`
- `static boolean isPrivate (int modifiers)`
- `static boolean isProtected (int modifiers)`
- `static boolean isPublic (int modifiers)`
- `static boolean isStatic(int modifiers)`
- `static boolean isStrict (int modifiers)`

- `static boolean isSynchronized (int modifiers)`
- `static boolean isVolatile (int modifiers)`

  Данные методы проверяют разряды числа `modifiers`, которые соответствуют модификаторам, определяемым именами методов.

Применение отражения для анализа объектов в ходе выполнения программы

В предыдущем разделе вы узнали, что для определения *имен* и *типов* полей любого объекта надо выполнить следующие действия.

- Получить соответствующий объект типа `Class`.
- Вызвать метод `getDeclaredFields()`.

В этом разделе мы попробуем определить *содержимое* поля. Разумеется, если имя и тип объекта известны при написании программы, это не составляет никакого труда. Однако отражение позволяет сделать это и для объектов, которые не известны на этапе компиляции.

Ключевым в этом процессе является метод `get()` класса `Field`. Если объект `f` имеет тип `Field` (например, если он получен в результате применения метода `getDeclaredFields()`), а объект `obj` относится к тому же классу, что и объект `f`, то вызов `f.get(obj)` вернет объект, значение которого будет текущим значением поля в объекте `obj`. Рассмотрим конкретный пример:

```
Employee harry = new Employee ("Harry Hacker", 3500, 10, 1, 1989);
Class cl = harry.getClass(0;
  //  Объект типа Class, представляющий класс Employee.
Field f = cl.getDeclaredField("name");
  // Имя поля для класса Employee.
Object V = f.get(harry);
  // Значение поля name объекта harry,
  // т.е. объект типа String "Harry Hacker".
```

Несмотря на то что код кажется несложным, при его выполнении возникнет проблема. Поскольку поле `name` объявлено как `private`, метод `get()` сгенерирует исключение `IllegalAccessException`. Метод `get()` можно применять только для общедоступных полей. Механизм безопасности языка Java позволяет определить, какие поля содержит объект, но не даст прочитать их содержимое, если эти поля недоступны.

По умолчанию механизм отражения выполняет правила доступа, установленные в языке Java. Однако, если программа не контролируется диспетчером защиты, эти правила можно обойти. Чтобы сделать это, вызовите метод `setAccessible()` для объектов `Field`, `Method` или `Constructor`, например:

```
f.setAccessible(true); // Теперь можно вызывать f.get(harry);
```

Метод `setAccessible()` относится к классу `AccessibleObject`, общему суперклассу классов `Field`, `Method` и `Constructor`. Он нужен для обеспечения работы отладчиков, поддержки постоянных хранилищ и выполнения других подобных действий. Позднее в этой главе мы применим его для создания настраиваемого метода `toString()`.

С методом get() связана еще одна проблема. Поле name имеет тип String, поэтому можно вернуть его значение как объект Object. Допустим, однако, что нам нужно определить значение поля salary. Оно имеет тип double, а в языке Java нет объектов числового типа. Чтобы решить эту проблему, можно использовать либо метод getDouble() класса Field, либо метод get(), который с помощью механизма отражения поместит поле в соответствующий интерфейсный класс, в данном случае — Double.

Разумеется, значение поля можно не только определять, но и задавать. Вызов f.set(obj, value) задает новое значение поля, представленного переменной f в объекте obj.

В листинге 5.6 показано, как создать обобщенный метод toString(), работающий с *любым* классом. Сначала он использует метод getDeclaredFields() для получения всех полей. Затем применяется метод setAccessible(), делающий все эти поля доступными. Далее определяется имя и значение каждого поля. В листинге 5.6 показан пример преобразования значений в строки путем рекурсивного вызова метода toString().

```
class ObjectAnalyzer
{
  public String toString(Object obj)
  {
      Class cl = obj.getClass();
      . . .
      String r = cl.getName();
      // Проверка полей класса и всех его суперклассов.
      do
      {
         r += "[";
         Field[] fields = cl.getDeclaredFields();
         AccessibleObject.setAccessible(fields, true);
         // Определение имен и значений всех полей.
         for (Field f : fields)
         {
            if (!Modifier.isStatic(f.getModifiers()))
            {
               if (!r.endsWith("[")) r += ","
               r += f.getName() + "=";
               try
               {
                  Object val = f.get(obj);
                  r += toString(val);
               }
               catch (Exception e) { e.printStackTrace(); }
            }
         }
         r += "]";
         cl = cl.getSuperclass();
      }
      while (cl != Object.class);
      return r;
   }
   . . .
}
```

Код, представленный в листинге 5.6, достаточно сложен для понимания. Циклические ссылки могут привести к бесконечной рекурсии, следовательно, объект `ObjectAnalyzer` должен отслеживать объекты, которые он уже просмотрел. Кроме того, для просмотра массивов необходим другой подход. Детали этого подхода мы рассмотрим в следующем разделе.

Для просмотра содержимого любого объекта можно воспользоваться методом `toString()`. Рассмотрим следующий вызов:

```
ArrayList<Integer> squares = new ArrayList<Integer>();
for (int i = 1; i <= 5; i++) squares.add(i * i);
   System.out.println(new ObjectAnalyzer().toString(squares));
```

В результате выполнения данных строк кода будет выведена следующая информация:

```
java.util.ArrayList[elementData=class
java.lang.Object[]{java.lang.Integer[value=1][][],
java.lang.Integer[value=4][][],java.lang.Integer[value=9][][],java.lang.In
teger[value=16][][],
java.lang.Integer[value=25][][],null,null,null,null,null},size=5][modCount
=5][][]
```

Универсальный метод `toString()` можно использовать для реализации метода `toString()` в собственных классах. Например, сделать это можно следующим образом:

```
public String toString()
{
   return new ObjectAnalyzer().toString(this);
}
```

Такой метод полезен при отладке программ.

Листинг 5.6. Содержимое файла `ObjectAnalyzerTest.java`

```
import java.lang.reflect.*;
import java.util.*;
import java.text.*;

public class ObjectAnalyzerTest
{
   public static void main(String[] args)
   {
      ArrayList<Integer> squares = new ArrayList<Integer>();
      for (int i = 1; i <= 5; i++) squares.add(i * i);
         System.out.println(new
            ObjectAnalyzer().toString(squares));
   }
}

class ObjectAnalyzer
{
   /**
      Преобразует объект в строку,
      в которой перечислены все поля.
      @param obj Объект
      @return Строка, в которой указаны имя объекта,
         имена всех полей и их значения.
```

```
   */
   public String toString(Object obj)
   {
      if (obj == null) return "null";
      if (visited.contains(obj)) return "...";
      visited.add(obj);
      Class cl = obj.getClass();
      if (cl == String.class) return (String) obj;
      if (cl.isArray())
      {
         String r = cl.getComponentType() + "[]{";
         for (int i = 0; i < Array.getLength(obj); i++)
         {
            if (i > 0) r += ",";
            Object val = Array.get(obj, i);
            if (cl.getComponentType().isPrimitive()) r += val;
            else r += toString(val);
         }
         return r + "}";
      }

      String r = cl.getName();
      // Проверка полей класса и всех его суперклассов.
      do
      {
         r += "[";
         Field[] fields = cl.getDeclaredFields();
         AccessibleObject.setAccessible(fields, true);
         // Определение имен и значений всех полей.
         for (Field f : fields)
         {
            if (!Modifier.isStatic(f.getModifiers()))
            {
               if (!r.endsWith("[")) r += ",";
               r += f.getName() + "=";
               try
               {
                  Class t = f.getType();
                  Object val = f.get(obj);
                  if (t.isPrimitive()) r += val;
                  else r += toString(val);
               }
               catch (Exception e) { e.printStackTrace(); }
            }
         }
         r += "]";
         cl = cl.getSuperclass();
      }
      while (cl != null);

      return r;
   }

   private ArrayList<Object> visited = new ArrayList<Object>();
}
```

API **java.lang.reflect.AccessibleObject 1.2**

- `void setAccessible(boolean flag)`

  Устанавливает признак доступности заданного объекта отражения. Значение `true` свидетельствует, что проверка доступа к полю отменена и закрытые поля теперь доступны.

- `boolean isAccessible()`

  Получает значение признака доступности заданного объекта отражения.

- `static void setAccessible(AccessibleObject[] array, boolean flag)`

  Удобный метод для установки признака доступности массива объектов.

API **java.lang.Class 1.1**

- `Field getField(String name)`
- `Field[] getFields()`

  Возвращает общедоступное поле с указанным именем или массив, содержащий все поля.

- `Field getDeclaredField(String name)`
- `Field[] getDeclaredFields()`

  Возвращает поле с указанным именем, объявленное в классе, или массив, содержащий все поля.

API **java.lang.reflect.Field 1.1**

- `Object get(Object obj)`

  Возвращает значение поля объекта `obj`, описываемого данным объектом `Field`.

- `void set(Object obj, Object newValue)`

  Устанавливает новое значение поля объекта `obj`, описываемого данным объектом `Field`.

Применение отражения для работы с универсальным массивом

Класс `Array` из пакета `java.lang.reflect` позволяет создавать массивы динамически. Например, используя метод `arrayCopy()`, о котором упоминалось в главе 3, можно увеличить массив, полностью сохранив его предыдущее содержимое.

Проблема, которую мы собираемся решить, весьма типична. Допустим, что у нас есть заполненный массив заданного типа и нам требуется увеличить его. Кроме того, предположим, что нам надоело каждый раз вручную создавать необходимый для этого код. Мы хотим написать обобщенный метод, позволяющий увеличивать размер массива, сохраняя его предыдущее содержание.

```
Employee[] a = new Employee[100];
...
// Массив полон.
a = (Employee[])arrayGrow(a);
```

Как написать такой метод? Нам поможет тот факт, что массив `Employee[]` можно преобразовать в массив `Object[]`. Звучит многообещающе. Итак, предпримем первую попытку создать обобщенный метод. Мы просто увеличим массив на 10%+10 элементов (поскольку одного увеличения на 10% для небольших массивов недостаточно).

```
static Object[] badArrayGrow(Object[] a) // Бесполезно.
{
   int newLength = a.length * 11 / 10 + 10;
   Object[] newArray = new Object[newLength];
   System.arraycopy(a, 0, newArray, 0 , a.length);
   return newArray;
}
```

Однако существует проблема, связанная с фактическим использованием полученного в результате массива. Этот массив имеет тип `Object[]`, поскольку он создан с помощью выражения

```
new Object[newLength]
```

Массив класса `Object[]` не может быть преобразован в массив класса `Employee[]`. При попытке сделать это возникнет исключение `ClassCast`. Проблема в том, что, как мы уже указывали, в ходе выполнения программы система запоминает первоначальный тип элементов массива, т.е. тип, указанный в операторе `new`. Можно временно преобразовать массив типа `Employee[]` в массив класса `Object[]` и обратно, но массив, изначально созданный как массив класса `Object[]`, преобразовать в массив класса `Employee[]` невозможно. Чтобы создать подходящий обобщенный метод, нужно научиться создавать новый массив, тип которого совпадал бы с типом исходного массива. Для этого нам нужны методы класса `Array` из пакета `java.lang.reflect`, особенно метод `newInstance()`, создающий новый массив. Тип элементов массива и требуемая длина должны передаваться в качестве параметров метода.

```
Object newArray = Array.newInstance(componentType, newLength);
```

Чтобы сделать это, нам нужно уметь определять длину и тип элементов нового массива.

Длину можно получить с помощью метода `Array.getLength()`. Статический метод `getLength()` класса `Array` возвращает длину любого массива. Чтобы определить тип элементов нового массива, нужно выполнить следующие действия.

1. Определить, какому классу принадлежит объект `a`.
2. Убедиться, что он действительно является массивом.
3. Использовать метод `getComponentType()` класса `Class` (определенный лишь для объектов, представляющих собой массивы) и получить требуемый тип массива.

Почему метод `getLength()` принадлежит классу `Array`, а `getComponentType()` — классу `Class`? Вряд ли это известно кому-либо, кроме разработчиков этих классов. Приходится принимать существующее распределение методов по классам.

В итоге мы получим следующий код:

```
static Object goodArrayGrow(Object a) // Правильный код.
{
  Class cl = a.getClass();
```

```
  if (!cl.isArray()) return null;
  Class componentType = cl.getComponentType();
  int length = Array.getLength(a);
  int newLength = length * 11/10 +10;
  Object newArray = Array.newInstance(componentType, newLength);
  System.arraycopy(a, 0, newArray, 0 , length);
  return newArray;
}
```

Заметим, что метод `arrayGrow()` можно применять для увеличения массива любого типа, а не только массива объектов.

```
int[] a = { 1, 2, 3, 4 };
a = (int[]) goodArrayGrow(a);
```

Для этого параметр метода `arrayGrow()` объявляется как объект `Object`, а не как массив объектов `Object` (т.е. `Object[]`). Массив типа `int[]` можно преобразовать в объект типа `Object`, но не в массив, состоящий из экемпляров класса `Object`!

В листинге 5.7 показано, как следует увеличивать размеры массивов разных типов. Отметим, что приведение типа значения, возвращаемого методом `badArrayGrow()`, вызовет исключение.

Листинг 5.7. Содержимое файла `ArrayGrowTest.java`

```
import java.lang.reflect.*;
import java.util.*;

public class ArrayGrowTest
{
   public static void main(String[] args)
   {
      int[] a = { 1, 2, 3 };
      a = (int[]) goodArrayGrow(a);
      arrayPrint(a);

      String[] b = { "Tom", "Dick", "Harry" };
      b = (String[]) goodArrayGrow(b);
      arrayPrint(b);

      System.out.println(
         "The following call will generate an exception.");
      b = (String[]) badArrayGrow(b);
   }

   /**
      Этот метод пытается увеличить размер массива, создавая
      новый массив и копируя туда все элементы старого.
      @param a массив, подлежащий увеличению
      @return увеличенный массив, содержащий все элементы a.
         Возвращаемый массив имеет тип Object[],
      который не совпадает с типом массива a.
   */
   static Object[] badArrayGrow(Object[] a)
   {
      int newLength = a.length * 11 / 10 + 10;
```

```
        Object[] newArray = new Object[newLength];
        System.arraycopy(a, 0, newArray, 0, a.length);
        return newArray;
    }

    /**
        Этот метод увеличивает размер массива, создавая новый
        массив того же типа и копируя туда все имеющиеся элементы.
        @param a массив, подлежащий увеличению. Может состоять из
          объектов или значений простых типов.
          return увеличенный массив, содержащий все элементы массива a.
    */
    static Object goodArrayGrow(Object a)
    {
        Class cl = a.getClass();
        if (!cl.isArray()) return null;
        Class componentType = cl.getComponentType();
        int length = Array.getLength(a);
        int newLength = length * 11 / 10 + 10;

        Object newArray =
            Array.newInstance(componentType, newLength);
        System.arraycopy(a, 0, newArray, 0, length);
        return newArray;
    }

    /**
      Удобный метод вывода всех элементов массива.
      @param a Массив для вывода. Может состоять из
        объектов или значений простых типов.
    */
    static void arrayPrint(Object a)
    {
        Class cl = a.getClass();
        if (!cl.isArray()) return;
        Class componentType = cl.getComponentType();
        int length = Array.getLength(a);
        System.out.print(componentType.getName()
            + "[" + length + "] = { ");
        for (int i = 0; i < Array.getLength(a); i++)
            System.out.print(Array.get(a, i) + " ");
        System.out.println("}");
    }
}
```

API **java.lang.reflect.Array 1.1**

- `static Object get(Object array, int index)`
- `static xxx getXxx(Object array, int index)`

  (Символы *xxx* обозначают простые типы `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`.) Эти методы возвращают значение элемента указанного массива, имеющего указанный индекс.

- `static void set(Object array, int index, Object newValue)`

- `static set`*`Xxx`*`(Object array, int index,` *`xxx`* `newValue)`

  (Символы *xxx* обозначают простые типы `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`.) Эти методы присваивают новое значение элементу указанного массива, имеющему указанный индекс.

- `static int getLength(Object array)`

  Возвращает длину указанного массива.

- `static Object newInstance(Class componentType, int length)`
- `static Object newInstance(Class componentType, int[] lengths)`

  Возвращает новый массив, состоящий из заданных компонентов и имеющий указанную размерность.

Указатели на методы!

На первый взгляд кажется, что в языке Java не предусмотрены указатели на методы, т.е. нет способа передать одному методу адрес другого, чтобы затем метод обратился по заданному адресу. Разработчики языка Java заявили, что указатели на методы небезопасны и часто порождают ошибки, а действия, для которых часто применяются такие указатели, удобно выполнять с помощью *интерфейсов* (они будут обсуждаться в следующей главе). Однако на самом деле, работая с JDK 1.1 и более поздними версиями, можно создавать указатели на методы. Такая возможность является побочным (возможно, случайным) продуктом механизма отражения.

Среди нестандартных расширений языка Java, которые компания Microsoft внесла в свой язык J++ (и в появившийся затем C#), есть тип указателей на методы, отличающийся от класса `Method`, обсуждаемого в этом разделе. Эта технология называется делегированием. Однако внутренние классы, которые рассматриваются в следующей главе, — более полезный и универсальный механизм, чем делегирование.

Чтобы понять, как работают указатели на методы, вспомните, что поле объекта можно проверить с помощью метода `get()` класса `Field`. Аналогично класс `Method` содержит метод `invoke()`, позволяющий вызывать метод, содержащийся в текущем объекте этого класса.

```
Object invoke(Object obj, Object... args)
```

Первый параметр является неявным, а остальные объекты представляют собой явные параметры. (До появления JDK 5.0 приходилось передавать методу массив объектов или, если явные параметры отсутствовали, задавать значение `null`.)

Если метод статический, первый параметр игнорируется; вместо него можно указать `null`.

Например, если `m1` представляет метод `getName()` класса `Employee`, можно использовать следующее выражение:

```
String n = (String) m1.invoke(harry);
```

Как это уже было с методами `get()` и `set()` класса `Field`, если параметр или возвращаемое значение не являются объектами какого-либо класса, а представляет собой простой тип, возникает проблема. При работе с версиями JDK, предшествующими 5.0, нужно было сначала преобразовать все базовые типы в соответствующие типы интерфейсных клас-

сов. С появлением JDK 5.0 можно полагаться на автоматическое преобразование простых типов в классы.

Если возвращаемое значение относится к простому типу, метод `invoke()` возвращает объект интерфейсного класса. Допустим, что `m2` представляет метод `raiseSalary()` класса `Employee`. В результате выполнения метода будет возвращен объект `Double`, после чего вы сможете выполнить приведение типов. В JDK 5.0 преобразование произойдет автоматически.

```
double s = (Double) m2.invoke(harry);
```

Как получить объект `Method`? Можно, конечно, вызвать метод `getDeclaredMethods()` и выполнить поиск среди возвращаемого массива объектов типа `Method`. Кроме того, можно вызвать метод `getMethod()` класса `Class`. Его действие можно сравнить с методом `getField()`, получающим строку с указанием имени поля и возвращающим объект `Field`. Однако методов с одним и тем же именем может быть несколько, и следует тщательно выбирать требуемый. По этой причине нужно предусмотреть массив, содержащий правильные типы параметров. Сигнатура метода `getMethod()` выглядит следующим образом:

```
Method getMethod(String name, Class... parameterTypes)
```

Рассмотрим, например, как можно получить указатели на методы `getName()` и `raiseSalary()` класса `Employee`.

```
Method m1 = Employee.class.getMethod("getName");
Method m2 = Employee.class.getMethod("raiseSalary", double.class);
```

До появления JDK 5.0 приходилось размещать объекты `Class` в массиве, либо, при отсутствии параметров, указывать значение `null`.

Теперь, выяснив правила использования объектов `Method`, рассмотрим, как они работают. В листинге 5.8 приведена программа, которая выводит таблицу значений математической функции, например `Math.sqrt` или `Math.sin`. Результат работы программы выглядят следующим образом:

```
public static native double java.lang.Math.sqrt(double)
     1.0000 |     1.0000
     2.0000 |     1.4142
     3.0000 |     1.7321
     4.0000 |     2.0000
     5.0000 |     2.2361
     6.0000 |     2.4495
     7.0000 |     2.6458
     8.0000 |     2.8284
     9.0000 |     3.0000
    10.0000 |     3.1623
```

Разумеется, код, осуществляющий вывод таблицы на экран, не зависит от конкретной функции.

```
double dx = (to - from) / (n - 1);
for (double x = from; x <= to; x += dx)
{
   double y = (Double) f.invoke(null, x);
```

```
      System.out.printf("%10.4f | %10.4f%n" + y, x, y);
   }
```

Здесь `f` — это объект `Method`. Поскольку мы вызываем статический метод, то в качестве первого `invoke()` передается `null`,

Для того чтобы вывести таблицу со значениями функции `Math.sqrt`, мы используем следующую строку кода:

```
Math.class.getMethod("sqrt", double.class)
```

При вызове `getMethod()` передается имя метода `sqrt` класса `Math` и параметр типа `double`.

В листинге 5.8 приведен полный код универсальной программы вывода значений функции.

Листинг 5.8. Содержимое файла `MethodPointerTest.java`

```
import java.lang.reflect.*;

public class MethodPointerTest
{
   public static void main(String[] args) throws Exception
   {
      // Получение указателей на методы square и sqrt.
      Method square = MethodPointerTest.class.getMethod("square",
         double.class);
      Method sqrt = Math.class.getMethod("sqrt", double.class);

      // Вывод значений x и y в виде таблицы.

      printTable(1, 10, 10, square);
      printTable(1, 10, 10, sqrt);
   }

   /**
      Возвращает квадрат числа.
      @param x Число
      @return Квадрат числа
   */
   public static double square(double x)
   {
      return x * x;
   }

   /**
      Выводит таблицу значений x и y метода.
      @param from Нижняя граница значений x
      @param to Верхняя граница значений x
      @param n Количество строк в таблице
      @param f Метод, получающий и возвращающий значения
            типа double
   */
   public static void printTable(double from, double to,
      int n, Method f)
   {
```

```
        // Вывод метода в качестве заголовка таблицы.
        System.out.println(f);

        double dx = (to - from) / (n - 1);

        for (double x = from; x <= to; x += dx)
        {
           try
           {
              double y = (Double) f.invoke(null, x);
              System.out.printf("%10.4f | %10.4f%n", x, y);
           }
           catch (Exception e) { e.printStackTrace(); }
        }
     }
}
```

Таким образом, с помощью объектов `Method` можно делать то же, что и посредством указателей на функции в языке C++ (или делегирования в C#). Как и в языке C++, такой стиль программирования обычно неудобен и часто приводит к ошибкам. Что случится, если мы вызовем метод с неправильными параметрами? Метод `invoke()` сгенерирует исключение.

Кроме того, и параметры, и возвращаемое значение метода `invoke()` обязательно должны иметь тип `Object`. Это значит, что нам придется осуществлять приведение типов, в результате компилятор не станет проверять программу. Следовательно, ошибки проявятся только при ее тестировании, когда исправить их будет значительно сложнее. Более того, программа, использующая механизм отражения для получения указателей на методы, работает гораздо медленнее, чем программа, непосредственно вызывающая эти методы.

По этой причине мы рекомендуем использовать объекты `Method` только в случае крайней необходимости. Намного лучше применять интерфейсы и внутренние классы (тема следующей главы). В частности, вслед за разработчиками языка Java мы повторим, что использовать объекты `Method` для обратного вызова функций не следует. Для этой цели хорошо подходят интерфейсы (см. следующую главу), позволяющие создавать программы, которые работают намного быстрее и надежнее.

java.lang.reflect.Method 1.1

- `public Object invoke(Object implicitParameter, Object[] explicit(parameters)`

  Вызывает метод, описанный в объекте, передавая ему заданные параметры и возвращая значение, вычисленное этим методом. Для статических методов в *качестве неявного параметра передается ссылка* `null`. *Простые типы следует* передавать только в виде интерфейсных классов. Значения простых типов должны извлекаться из интерфейсных классов.

Нумерованные типы

В главе 3 было сказано, что начиная с JDK 5.0 поддерживаются нумерованные типы. Пример соответствующего выражения приведен ниже.

```
public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

Тип, объявленный таким образом, на самом деле представляет собой класс. Возможно существование всего четырех экземпляров указанного класса. Другие объекты создать нельзя.

Таким образом, вам не обязательно использовать метод `equals()`, чтобы проверять эквивалентность нумерованных значений. Для этой цели вполне подойдет оператор ==.

При желании вы можете добавлять к нумерованным типам конструкторы, методы и поля. Очевидно, что конструкторы могут вызываться только при создании нумерованных констант. Ниже приведен пример кода класса.

```
enum Size
{
   SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

   private Size(String abbreviation) { this.abbreviation =
      abbreviation; }
   public String getAbbreviation() { return abbreviation; }

   private String abbreviation;
}
```

Все нумерованные типы реализуются с помощью подклассов класса `Enum`. Они наследуют от этого класса ряд методов. Наиболее часто используется метод `toString()`, возвращающий имя нумерованной константы. Например, метод `Size.SMALL.toString()` вернет строку "SMALL".

Статический метод `valueOf()` выполняет действия, противоположные методу `toString()`. Например, в результате выполнения приведенного ниже выражения переменной `s` будет присвоено значение `Size.SMALL`.

```
Size s = (Size) Enum.valueOf(Size.class, "SMALL");
```

Каждый нумерованный тип содержит статический метод `values()`, который возвращает массив всех нумерованных значений.

```
Size[] values = Size.values();
```

Работу с нумерованными типами демонстрирует пример, приведенный в листинге 5.9.

В отличие от класса `Class`, класс `Enum` допускает указание типа. Например, нумерованный тип `Size` на самом деле преобразуется в `Enum<Size>`. В данном разделе мы не рассматривали эту возможность.

Листинг 5.9. Содержимое файла `EnumTest.java`

```
import java.util.*;

public class EnumTest
{
```

```
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print(
         "Enter a size: (SMALL, MEDIUM, LARGE, EXTRA_LARGE) ");
      String input = in.next().toUpperCase();
      Size size = Enum.valueOf(Size.class, input);
      System.out.println("size=" + size);
      System.out.println("abbreviation=" +
         size.getAbbreviation());
      if (size == Size.EXTRA_LARGE)
         System.out.println(
            "Good job--you paid attention to the _.");
   }
}

enum Size
{
   SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

   private Size(String abbreviation)
      { this.abbreviation = abbreviation; }
   public String getAbbreviation()
      { return abbreviation; }

   private String abbreviation;
}
```

java.lang.Enum 5.0

- `static Enum valueOf(Class enumClass, String name)`

  Возвращает нумерованную константу указанного класса с заданным именем.

- `String toString()`

  Возвращает имя нумерованной константы.

Как применять наследование

Заканчивая данную главу, предложим несколько советов по правильному применению очень полезного механизма наследования.

1. *Размещайте общие операции и поля в суперклассе.*

   По этой причине мы поместили поле `name` в класс `Person` и не стали повторять его в классах `Employee` и `Student`.

2. *Старайтесь не использовать защищенные поля.*

   Некоторые разработчики полагают, что следует "на всякий случай" объявлять большинство полей защищенными, чтобы по мере необходимости подклассы могли к ним обратиться. Однако есть две причины, по которым этот механизм не гарантирует достаточной защиты. Во-первых, множество подклассов неограниченно — любой программист может создать подкласс вашего класса и написать программу, получающую непосредственный доступ к защищенным полям экземпляра, нарушая инкапсуляцию. Во-вторых, в языке Java к защищен-

ным полям имеют доступ все классы, размещенные в том же пакете, независимо от того, являются они подклассами данного класса или нет.

Однако специализированные методы, которые переопределяются в подклассах, защищенными объявлять полезно. Ярким примером служит метод `clone()`.

3. *Используйте наследование для моделирования отношения "является".*
Наследование позволяет экономить время и усилия при разработке программ, поэтому некоторые программисты им злоупотребляют. Допустим, нам нужен класс `Contractor`. Сотрудник, нанимаемый по контракту, имеет имя и дату заключения договора, однако у него нет оклада. У него почасовая оплата, причем он работает не так давно, чтобы возникла необходимость повышать оплату его труда. Ниже показано, как можно сделать класс `Contractor` подклассом класса `Employee`, добавив поле `hourlyWage`.

```
class Contractor extends Employee
{
   ...
   private double hourlyWage;
}
```

Эта идея *не* слишком удачна, поскольку при этом получается, что каждый сотрудник, нанятый по контракту, имеет и оклад, и почасовую оплату. Если вы попробуете реализовать методы для распечатки чеков или ведомостей, сразу проявится недостаток такого подхода. Программа, которую вам придется написать, будет гораздо длиннее той, которую вы могли бы создать, не прибегая к неоправданному наследованию.

Отношение "сотрудник с почасовой оплатой/постоянный сотрудник" не удовлетворяет отношению "является". Сотрудники, *нанятые по контракту*, не являются разновидностью *постоянных* сотрудников.

4. *Не используйте наследование, если не все наследуемые методы имеют смысл.* Допустим, что нам нужно написать класс `Holiday`. Разумеется, праздники — это разновидность календарных дней, а дни можно представить в виде объектов `GregorianCalendar`, поэтому можно применять наследование.

```
class Holiday extends GregorianCalendar(...)
```

К сожалению, множество праздников при наследовании не закрыто. Среди открытых методов класса `GregorianCalendar` есть метод `add()`, который может превратить праздники в будни.

```
Holiday christmas;

cristmas.add(GregorianCalendar.DAY, 12);
```

Итак, наследование в этом случае не подходит.

5. *Переопределяя метод, не изменяйте ожидаемое поведение класса.*
При переопределении методов необходимо не только соблюдать синтаксис, но, что гораздо важнее, учитывать поведение класса. В этом компилятор не поможет вам; он не в состоянии проверить, оправдано ли переопределение метода. Предположим, что вы захотели устранить проблему с методом `add()` в классе `Holiday`, переопределив этот метод так, чтобы он, например, не выполнял ника-

ких действий либо возвращал следующий праздничный день. Однако такое переопределение не оправдано. При выполнении приведенной ниже последовательности команд пользователь вправе ожидать определенного результата, независимо от того, является ли x экземпляром `GregorianCalendar` или `Holiday`.

```
int d1 = x.get(Calendar.DAY_OF_MONTH);
x.add(Calendar.DAY_OF_MONTH, 1);
int d2 = x.get(Calendar.DAY_OF_MONTH);
System.out.println(d2 - d1);
```

Очевидно, что приведенный пример несколько надуман. Разные пользователи посчитают естественным различное поведение программы. Так, например, можно услышать мнение, что в методе `Manager.equals()` не должна учитываться премия, поскольку метод `Employee.equals()` игнорирует ее. Подобные споры могут длиться бесконечно и не дать никакого результата. Принимая конкретное решение, следует руководствоваться целями, для которых создается программа.

6. *Используйте полиморфизм, а не информацию о типе.*

   Вспомните о полиморфизме, как только увидите код, имеющий следующий вид:

   ```
   if (x имеет_тип 1)
       действие_1(x);
   else if (x имеет_тип 2)
       действие_2(x);
   ```

   Носят ли *действие_1* и *действие_2* общий характер? Если да, поместите соответствующие методы в общий суперкласс или интерфейс обоих типов. Тогда можно просто вызвать приведенный ниже метод и с помощью механизма динамического связывания, присущего полиморфизму, выполнить правильное действие.

   ```
   x.действие();
   ```

   Код, использующий полиморфизм или реализацию интерфейса, намного легче поддерживать и расширять, чем код, использующий многочисленные проверки типов.

7. *Не злоупотребляйте механизмом отражения.*

   Механизм отражения позволяет создавать программы с высоким уровнем абстракции, определяющие поля и методы в ходе выполнения программы. Эта возможность чрезвычайно полезна для системного программирования, но для прикладного — практически не нужна. Отражение — очень хрупкий механизм, поскольку компилятор не может помочь программисту обнаружить ошибки. Все ошибки проявляются во время выполнения программы и приводят к возникновению исключений.

Глава 6

Интерфейсы и внутренние классы

В этой главе...

- Интерфейсы
- Клонирование объектов
- Интерфейсы и обратный вызов
- Внутренние классы
- Proxy-классы

Теперь вы знаете все основные инструменты, предназначенные для объектно-ориентированного программирования на языке Java. В этой главе мы рассмотрим еще два широко распространенных способа программирования. Несмотря на то что эти способы далеко не очевидны, владеть ими должен каждый профессиональный программист.

Первый из них называется *программирование интерфейсов* (interfaces). Этот способ позволяет указывать, *что* должны делать классы, не описывая, *как* они это должны делать. Классы могут *реализовывать* один или несколько интерфейсов. Если возникает необходимость использовать интерфейс, применяются объекты этих классов. Изучив интерфейсы, мы рассмотрим клонирование объектов (которое иногда называется *"глубоким" копированием*). Клоном объекта называется новый объект, имеющий точно такое же состояние.

Далее мы перейдем к механизму, основанному на применении *внутренних классов* (inner class). С технической точки зрения внутренние классы довольно сложны – они определяются внутри других классов, а их методы имеют доступ к полям окружающего класса. Внутренние классы полезны при разработке совокупности взаимосвязанных классов. В частности, они позволяют создавать лаконичные, высокопрофессиональные коды для реализации событийно-ориентированного графического пользовательского интерфейса.

Глава завершается обсуждением proxy-объектов, реализующих произвольные интерфейсы. *Proxy-объекты представляют собой специализированные конструкции*, полезные для создания инструментов системного программирования. Начинающие *программисты могут спокойно пропустить этот раздел.*

Интерфейсы

Интерфейс в языке Java не является классом. Он представляет собой множество *требований* (requirements), предъявляемых к соответствующему классу.

Обычно разработчик, который собирается использовать работу другого программиста для решения какой-либо задачи, заявляет: "Если ваш класс будет соответствовать определенному интерфейсу, я смогу выполнить свою работу". Рассмотрим конкретный пример. Метод `sort` класса `Array` позволяет упорядочить массив объектов при одном условии: объекты должны принадлежать классам, реализующим интерфейс `Comparable`.

Вот как выглядит этот интерфейс:

```
public interface Comparable
{
   int compareTo (Object other);
}
```

Это значит, что любой класс, реализующий интерфейс `Comparable`, должен иметь метод `compareTo()`, получающий параметр `Object` и возвращающий целое число.

В JDK 5.0 появилась возможность указывать в интерфейсе `Comparable` тип данных.

```
public interface Comparable<T>
{
   int compareTo(T other); // Параметр должен принадлежать типу T
}
```

Например, если класс реализует интерфейс `Comparable<Employee>`, он должен содержать следующий метод:

```
int compareTo(Employee other);
```

При желании вы по-прежнему можете использовать "низкоуровневый" тип `Comparable`, но при этом вам придется явным образом приводить параметр метода `compareTo()` к требуемому типу.

Все методы интерфейса автоматически считаются общедоступными, поэтому, объявляя метод в интерфейсе, не обязательно указывать модификатор `public`.

Разумеется, существует и неявное требование: метод `compareTo()` должен на самом деле уметь сравнивать два объекта и возвращать признак того, что один из них больше другого. Предполагается, что метод возвращает отрицательное значение, если объект `x` меньше объекта `y`, нуль — если они равны, и положительное число — в противном случае.

В составе интерфейсов может быть объявлен один или несколько методов. Как вы узнаете далее, с помощью интерфейсов можно также объявлять константы. Однако еще важнее тот факт, что интерфейсы *не могут* реализовываться в виде объектов. В интерфейсах нет ни полей, ни тела методов; они содержатся в классах, реализующих соответствующие интерфейсы. Таким образом, может показаться, что интерфейс похож на абстрактный класс, лишенный каких-либо полей. Однако между двумя этими понятиями есть существенное различие, и позднее мы подробно проанализируем его.

Теперь предположим, что мы собираемся использовать метод `sort()` класса `Array` для сортировки объектов класса `Employee`. В этом случае класс `Employee` должен *реализовать* интерфейс `Comparable`.

Для того чтобы класс реализовал интерфейс, нужно выполнить два действия.

1. Объявить, что класс реализует интерфейс.
2. Определить в классе все методы, указанные в интерфейсе.

Чтобы объявить, что класс реализует интерфейс, используется ключевое слово `implements`:

```
class Employee implements Comparable
```

Разумеется, теперь нужно реализовать метод `compareTo()`. Например, сравним зарплату сотрудников. Метод `compareTo()` будет возвращать -1, если зарплата первого сотрудника меньше зарплаты второго, 0, если они равны, и 1 в противном случае.

```
public int compareTo(Object otherObject)
{
   Employee other = (Employee) otherObject;
   if (salary < other.salary) return -1;
   if (salary > other.salary) return 1;
   return 0;
}
```

В составе интерфейса для метода `compareTo()` не указывается модификатор `public`, поскольку все методы интерфейса автоматически являются общедоступными. Однако при реализации интерфейса этот модификатор должен быть указан. В противном случае компилятор предположит, что область видимости этого метода ограничивается пакетом, и выдаст предупреждение, что вы пытаетесь изменить уровень доступа.

Начиная с JDK 5.0 можно использовать более изящное решение, реализовав интерфейс `Comparable<Employee>`.

```
class Employee implements Comparable<Employee>
{
   public int compareTo(Employee other)
   {
      if (salary < other.salary) return -1;
      if (salary > other.salary) return 1;
      return 0;
   }
   . . .
}
```

Как видите, теперь не приходится приводить `Object` к требуемому типу.

Теперь ясно, что нужно сделать, чтобы класс смог сортировать объекты, — реализовать в нем метод `compareTo()`. Это вполне резонно. Должен существовать какой-то способ использовать метод `sort()` для сравнения объектов. Однако почему нельзя в классе `Employee` просто предусмотреть метод `compareTo()`, не реализуя интерфейс `Comparable`?

Метод `compareTo()` интерфейса `Comparable` возвращает целое число. Если объекты не совпадают, возвращается положительное или отрицательное значение, не равное нулю. Возможность использования в качестве признака неравенства любое число, отличное от нуля, может оказаться полезной при сравнении целочисленных полей. Допустим, что каждый сотрудник имеет уникальный номер `id`. В этом случае можно вернуть число `id - other.id`. Это значение будет отрицательным, если номер первого сотрудника меньше номера второго, 0, если они равны, и положительное в противном случае. Учтите только, что диапазон изменения целых чисел должен быть достаточно малым, чтобы вычитание не привело к переполнению. Если известно, что идентификационный номер лежит в пределах от 0 до `(Integer.MAX_VALUE - 1) / 2`, то вы смело можете применять описанный выше способ.

Разумеется, этот прием с вычитанием не работает для чисел с плавающей точкой. Разность `salary - other.salary` может округляться до нуля, если зарплаты очень близки, но не идентичны.

Причина заключается в том, что язык Java является *строго типизированным*. Вызывая некий метод, компилятор должен убедиться, что этот метод действительно существует. В теле метода `sort()` могут находиться операторы наподобие следующих:

```
if (a[i].compareTo(a[j]) > 0)
{
   // Поменять местами объекты a[i] и a[j].
   ...
}
```

Компилятор должен знать, что объект `a[i]` действительно содержит метод `compareTo()`. Если переменная `a` представляет собой массив объектов, реализующих интерфейс `Comparable`, то существование такого метода гарантируется, поскольку каждый класс, реализующий указанный интерфейс, по определению должен содержать этот метод.

Может показаться, что метод `sort()` класса Array работает только с массивами типа `Comparable[]` и что компилятор выдаст предупреждение, как только обнаружит вызов метода `sort()` для массива, элементы которого не реализуют указанный интерфейс. Увы, это не так. Вместо этого метод `sort()` получает на вход массивы типа `Object[]` и выполняет приведение типов.

```
// Фрагмент кода из стандартной библиотеки
if (((Comparable)a[i]).compareTo(a[i] > 0)
{
   // Поменять местами объекты a[i] и a[j].
   ...
}
```

Если объект `a[i]` не принадлежит классу, реализующему интерфейс `Comparable`, виртуальная машина генерирует исключение.

В листинге 6.1 приведен код программы сортировки массива, состоящего из объектов класса `Employee`.

Листинг 6.1. Содержимое файла **EmployeeSortTest.java**

```
import java.util.*;

public class EmployeeSortTest
{
   public static void main(String[] args)
   {
      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Harry Hacker", 35000);
      staff[1] = new Employee("Carl Cracker", 75000);
      staff[2] = new Employee("Tony Tester", 38000);

      Arrays.sort(staff);

      // Вывод информации о всех объектах Employee.
      for (Employee e : staff)
         System.out.println("name=" + e.getName() +
            ",salary=" + e.getSalary());
   }
}

class Employee implements Comparable<Employee>
{
   public Employee(String n, double s)
   {
      name = n;
      salary = s;
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   /**
      Сравнивает зарплату сотрудников.
      @param otherObject другой объект класса Employee
      @return отрицательное число, если зарплата данного
         сотрудника меньше, чем зарплата сотрудника,
         представленного объектом otherObject; 0,
         если их зарплаты равны, и положительное число
         в противном случае.
   */
```

```
   public int compareTo(Employee other)
   {
      if (salary < other.salary) return -1;
      if (salary > other.salary) return 1;
      return 0;
   }

   private String name;
   private double salary;
}
```

java.lang.Comparable 1.0

- `int compareTo(Object otherObject)`

  Сравнивает текущий объект с объектом `otherObject` и возвращает отрицательное целое число, если текущий объект меньше, чем объект `otherObject`, нуль, если они равны между собой, и положительное число в противном случае.

java.lang.Comparable<T> 5.0

- `int compareTo(T other)`

  Сравнивает текущий объект с объектом `other` и возвращает отрицательное целое число, если текущий объект меньше, чем объект `other`, нуль, если они равны между собой, и положительное число в противном случае.

java.util.Arrays 1.2

- `static void sort(Object[] a)`

  Сортирует элементы массива `a` методом слияния. Все элементы массива должны соответствовать классам, реализующим интерфейс `Comparable`, и быть совместимыми друг с другом.

В языке Java есть стандартное требование: "Автор реализации метода должен гарантировать, что для всех объектов `x` и `y` выполняется условие `sgn(x.compareTo(y) = -sgn(y.compareTo(x)`. (Следовательно, если вызов `y.compareTo(x)` генерирует исключение, значит, то же самое должен делать `x.compareTo(y)`.)" Здесь `sgn` означает функцию "сигнум", т.е. *знак* числа: `sgn(n)` равна –1, если число `n` отрицательно, 0, если число `n` равно нулю, и 1, если число `n` положительное. Другими словами, если поменять явный и неявный параметры метода `compareTo()` местами, знак возвращаемого числа (но не обязательно его фактическое значение) также должен измениться на противоположный.

Что касается метода `equals()`, при наследовании может возникнуть проблема.

Поскольку класс `Manager` расширяет класс `Employee`, он реализует интерфейс `Comparable<Employee>`, а не `Comparable<Manager>`. Если разработчик класса `Manager` примет решение переопределить метод `compareTo()`, он должен предусмотреть сравнение менеджеров с обычными сотрудниками. Можно привести тип `Employee` к типу `Manager`.

```
class Manager extends Employee
{
   public int compareTo(Employee other)
   {
      Manager otherManager = (Manager) other;     // Нельзя
      . . .
   }
   . . .
}
```

Однако такой подход не даст желаемого результата. Если объект `x` относится к классу `Employee`, а объект `y` — к классу `Manager`, то вызов `x.compareTo(y)` не приведет к возникновению исключения — он просто сравнит `x` и `y` как объекты класса `Employee`. Однако обратный вызов `y.compareTo(x)` сгенерирует исключение `ClassCastException`.

Аналогичная ситуация возникает при программировании метода `equals()`, которая обсуждалась в главе 5.

Если подклассы имеют различные критерии сравнения, то считаем, что сравнение недопустимо. Таким образом, каждый метод `compareTo()` должен начинаться со следующей проверки:

```
if (getClass() != other.getClass()) throw new ClassCastException();
```

Если существует общий алгоритм сравнения объектов подклассов, надо реализовать в суперклассе единый метод `compareTo()` и объявить его как `final`.

Предположим, например, что на предприятии, независимо от заработной платы, менеджеры считаются выше рядовых сотрудников. Как быть с другими подклассами, например `Executive` и `Secretary`? Если вы хотите учредить нечто вроде табеля о рангах, введите в состав класса `Employee` метод `rank()`. Тогда в каждом подклассе метод `rank()` должен переопределяться и результаты его работы учитываться при выполнении метода `compareTo()`.

Свойства интерфейсов

Интерфейсы – это не классы. В частности, с помощью оператора `new` невозможно создать экземпляр интерфейса.

```
x = new Comparable(...); // Неправильно.
```

Но несмотря на это, можно объявлять переменные, ссылающиеся на интерфейсы.

```
Comparable x; // Правильно.
```

Реально такая переменная ссылается на объект класса, реализующего данный интерфейс.

```
x = new Employee(...); // Правильно, если класс Employee
                       // реализует интерфейс Comparable.
```

Как известно, оператор `instanceof` проверяет, принадлежит ли объект заданному классу. Однако с помощью этого оператора можно также проверять, реализует ли объект заданный интерфейс.

```
if (anObject instanceof Comparable) { ... }
```

Аналогично классам, интерфейсы также могут образовывать иерархию наследования. Это позволяет создавать цепочки интерфейсов в направлении от более абстрактных к более специализированным. Допустим, что у нас есть интерфейс Moveable.

```
public interface Moveable
{
   void move(double x, double y);
}
```

Мы можем создать интерфейс Powered, расширяющий интерфейс Moveable.

```
public interface Powered extends Moveable
{
   double milesPerGallon();
}
```

Хотя в интерфейсе может не быть ни полей, ни статических методов, в нем можно объявлять константы.

```
public interface Powered extends Moveable
{
   double milesPerGallon();

   // Общедоступная статическая константа.
   double SPEED_LIMIT = 95;
}
```

Напомним, что все методы в интерфейсе автоматически считаются общедоступными. Аналогично, при объявлении константы всегда предполагаются модификаторы public static final.

Если при объявлении метода в составе интерфейса вы укажете ключевое слово public, а поле пометите как public static final, это не будет ошибкой. Некоторые программисты поступают так для того, чтобы коды их программ были более удобочитаемыми. Однако в спецификации языка Java рекомендуется не использовать лишние ключевые слова. (В данной книге мы следовали этим рекомендациям.)

В некоторых интерфейсах объявляются только константы и ни одного метода. Например, в стандартной библиотеке содержится интерфейс SwingConstants, определяющий константы NORTH, SOUTH, HORIZONTAL и т.д. Любой класс, реализующий интерфейс SwingContants, автоматически наследует эти константы. Его методы могут непосредственно ссылаться, например, на константу NORTH, не используя громоздкое обозначение SwingConstants.NORTH. Однако многие специалисты считают, что такое использование интерфейсов со временем изживет себя. Мы также не рекомендуем применять их с этой целью.

В языке Java любой класс может иметь только один суперкласс, но при этом любой класс может реализовывать *несколько* интерфейсов. Это позволяет максимально гибко определять поведение класса. Например, в языке Java есть очень важный интерфейс Cloneable. (Этот интерфейс детально изучается в следующем разделе.) Если некий класс реализует интерфейс Cloneable, то для создания точных копий его объектов можно применять метод clone() класса Object. Далее, допустим, что мы хотим

не только создавать клоны объектов нашего класса, но и сравнивать их между собой. Тогда нужно просто реализовать в классе оба интерфейса, `Clonable` и `Comparable`.

```
class Employee emplements Cloneable, Comparable
```

Для разделения имен интерфейсов, задающих свойства классов, используется запятая.

Интерфейсы и абстрактные классы

Если вы помните содержание раздела главы 5, посвященного абстрактным классам, у вас, возможно, возник вопрос, зачем разработчики языка Java создали механизм интерфейсов. Почему бы не сделать интерфейс `Comparable` абстрактным классом?

```
abstract class Comparable // Почему бы и нет?
{
   public abstract int compareTo(Object other);
}
```

В этом случае класс `Employee` мог бы просто расширять абстрактный класс и реализовывать метод `compareTo()`.

```
class Employee extends Comparable // Почему бы и нет?
{
   public int compareTo(Object other) { ... }
}
```

К сожалению, это породило бы массу проблем, связанных с использованием абстрактного базового класса для выражения обобщенного свойства. У класса может быть только один родитель. Допустим, что класс `Employee` уже является подклассом какого-нибудь другого класса, скажем, `Person`. Значит, еще одного родительского класса у него быть не может.

```
class Employee extends Person, Comparable // Ошибка!.
```

В то же время каждый класс может реализовывать сколько угодно интерфейсов.

```
class Employee extends Person implements Conparable // Правильно.
```

В других языках программирования, в частности в языке C++, классы могут иметь несколько суперклассов. Это свойство называется *множественным наследованием* (multiple inheritance). Разработчики языка Java решили не поддерживать множественное наследование, поскольку оно делает язык либо слишком сложным (как язык C++), либо менее эффективным (как язык Eiffel).

В то же время интерфейсы предоставляют большинство возможностей множественного наследования, не усложняя язык и не снижая эффективность.

В языке C++ есть множественное наследование. Это вызывает много сложностей, связанных с виртуальными базовыми классами, правилами доминирования и приведением типов указателей. Множественное наследование используют лишь немногие программисты, работающие на языке C++. Некоторые никогда его не применяют, остальные рекомендуют использовать множественное наследование только в "смешанном" виде. Это значит, что первичный базовый класс описывает родительский объект, а дополнительные базовые классы (так называемые *миксеры* (mix-ins)) описывают вспомогательные свойства. Этот стиль напоминает классы из языка Java, имеющие только один базовый класс и дополнительные интерфейсы. Однако в языке C++ миксеры могут добавлять некоторые свойства по умолчанию, в то время как интерфейсы в языке Java этого делать не могут.

Клонирование объектов

После создания копии переменной и оригинал, и копия представляют собой ссылки на один и тот же объект (рис. 6.1). Это значит, что изменение одной переменной повлечет изменение другой.

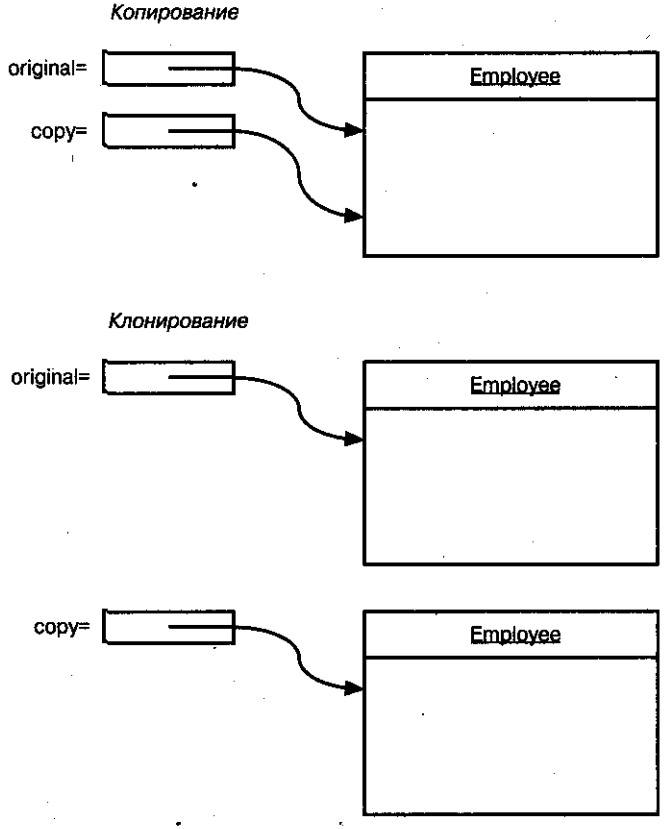


Рис. 6.1. Копирование и клонирование

```
Employee original = new Employee("John Public", 50000);
Employee copy = original;
copy.raiseSalary(10);  // Оригинал тоже изменился!
```

Если нужно, чтобы переменная `copy` представляла собой новый объект, в первый момент своего существования идентичный объекту `original`, но совершенно независимый от него, используйте метод `clone()`.

```
Employee copy = original.clone();
copy.raiseSalary(10); // OK! Оригинал не изменился!
```

Однако не все так просто. Метод `clone()` является защищенным (`protected`) методом класса `Object`, т.е. его нельзя вызвать непосредственно. Только класс `Employee` может клонировать объекты своего класса. У этого ограничения есть существенная причина. Подумайте о способе, которым класс `Object` может реализовать метод

`clone()`. Ему вообще ничего не известно об объекте, поэтому он может копировать лишь поля. Если все поля класса являются числовыми или имеют другой основной тип, их копирование выполняется прекрасно. Однако если объект содержит ссылку на подобъект, то и оригинал, и клонированные объекты будут совместно использовать одну и ту же информацию.

Чтобы проиллюстрировать это явление, рассмотрим класс `Employee`, который, начиная с главы 4, мы используем для демонстрации различных аспектов работы с объектами. На рис. 6.2 показано, что происходит, когда метод `clone()` класса `Object` применяется к объекту класса `Employee`. Как видим, операция клонирования по умолчанию является "поверхностной" – она не клонирует объекты, на которые есть ссылки внутри объектов.

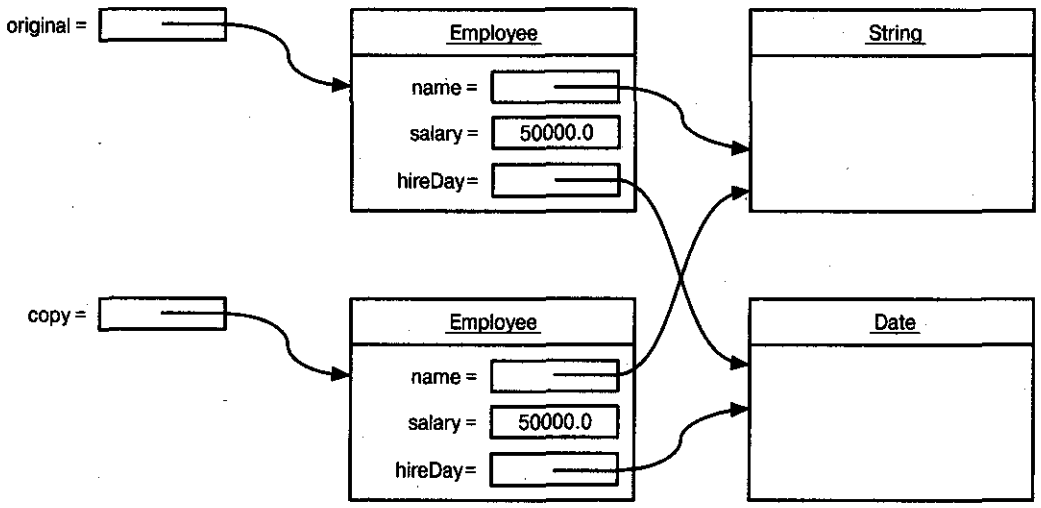


Рис. 6.2. "Поверхностная" копия

Хорошо ли это? Все зависит от конкретной ситуации. Если подобъект, используемый совместно как оригиналом, так и "поверхностным" клоном, является *неизменяемым* (immutable), это вполне безопасно. Такое происходит, если подобъект является экземпляром неизменяемого класса, например `String`. Подобъект также может оставаться постоянным на протяжении всей жизни содержащего его объекта, не подвергаясь воздействию модифицирующих методов либо методов, вычисляющих ссылку на него.

Достаточно часто, однако, подобъекты подвергаются изменениям, поэтому приходится переопределять метод `clone()`, чтобы выполнить *"глубокое" копирование* (deep copying), которое позволяет клонировать подобъекты наряду с содержащими их объектами. В нашем примере поле `hireDay` ссылается на экземпляр изменяемого класса `Date`.

Для каждого класса нужно ответить на следующие вопросы.

1. Достаточно ли метода `clone()` по умолчанию?
2. Можно ли дополнить метод `clone()`, используемый по умолчанию, применяя его к изменяемым объектам?
3. Следует ли применять метод `clone()`?

Считаем, что на третий вопрос получен положительный ответ. Для того чтобы программист мог использовать метод `clone()`, предлагаемый по умолчанию, либо модифицированный метод `clone()`, класс должен удовлетворять следующим требованиям.

1. Реализовать интерфейс `Cloneable`.
2. В нем должен быть переопределен метод `clone()` с модификатором доступа `public`.

Метод `clone()` в классе `Object` объявлен как `protected`, поэтому его нельзя просто вызвать с помощью выражения `anObject.clone()`. Однако разве недоступны защищенные методы для любого подкласса и не является ли каждый класс подклассом класса `Object`? Подкласс может вызвать защищенный метод `clone()` только для клонирования *своих собственных* объектов. Чтобы клонировать другие объекты, метод `clone()` следует переопределить и сделать общедоступным.

В данном случае интерфейс `Cloneable` используется не совсем привычным для вас образом. В частности, в нем не объявляется метод `clone()` — этот метод наследуется от класса `Object`. Интерфейс служит в качестве метки, указывающей на то, что в данном случае разработчик класса понимает, как выполняется процесс клонирования. Объекты настолько настороженно относятся к клонированию, что если объект требует выполнения данной операции, но не реализует интерфейс `Cloneable`, генерируется исключение.

Интерфейс `Cloneable` — один из немногих *интерфейсов-маркеров* (tagged interfaces) в языке Java. Напомним, что такие интерфейсы, как `Comparable`, должны гарантировать, что некий класс реализует конкретный метод или набор методов. Интерфейсы-маркеры не имеют методов; их единственное предназначение — позволить использовать оператор `instanceof` для проверки типа.

```
if (obj instanceof Cloneable) ...
```

Мы не рекомендуем использовать данную технологию в ваших программах.

Даже если применение метода `clone()` по умолчанию (поверхностное копирование) вполне допустимо, в любом случае нужно реализовать интерфейс `Cloneable`, переопределить метод `clone()`, сделав его открытым и вызвать метод `super.clone()`. Рассмотрим пример.

```
class Employee implements Cloneable
{
   // Область видимости расширяется до public,
   // и изменяется тип возвращаемого значения
   public Employee clone() throws CloneNotSupportedException
   {
      return super.clone();
   }
   . . .
}
```

До появления JDK 5.0 метод `clone()` всегда возвращал объект `Object`. Ковариантные возвращаемые типы, поддерживаемые JDK 5.0, позволяют указывать в методе `clone()` корректный тип возвращаемого значения.

Рассмотренный выше метод clone() не добавляет никаких новых функциональных возможностей к методу Object.clone(), реализующему "поверхностное" копирование. Чтобы реализовать "глубокое" копирование, надо приложить дополнительные усилия и организовать клонирование изменяемых полей экземпляра.

Ниже приведен пример метода clone(), выполняющего "глубокое" копирование.

```
class Employee implements Cloneable
{
   . . .
   public Object clone() throws CloneNotSupportedException
   {
      // Вызов метода Object.clone().
      Employee cloned = (Employee) super.clone();

      // Клонирование изменяемых полей.
      cloned.hireDay = (Date) hireDay.clone()

      return cloned;
   }
}
```

Метод clone() класса Object может генерировать исключение CloneNotSupportedException. Это происходит в том случае, если clone() вызывается для объекта, не реализующего интерфейс Cloneable. Поскольку и класс Employee, и класс Date реализуют этот интерфейс, исключение не генерируется. Однако компилятор на знает этого, поэтому нам надо объявить о возможном исключении следующим образом:

```
public Employee clone() throws CloneNotSupportedException
```

Не лучше ли было бы вместо этого предусмотреть обработку исключения?

```
public Employee clone()
{
   try
   {
      return super.clone();
   }
   catch (CloneNotSupportedException e) { return null; }
   // Это не произойдет, так как классы реализуют
   // интерфейс Cloneable
}
```

Такое решение вполне подходит для классов final. В других случаях уместнее прибегнуть к помощи ключевого слова throws. При этом у подкласса останется возможность сгенерировать исключение CloneNotSupportedException, в случае если он не сможет выполнить клонирование.

Клонируя подклассы, надо соблюдать осторожность. Например, если вы определите метод clone() в классе Employee, другие программисты смогут использовать его для клонирования объектов Manager. Сможет ли метод clone() из класса Employee справиться с данной задачей? Это зависит от набора полей, объявленных в классе Manager. В нашем случае проблемы не возникнут, поскольку поле bonus относится к простому типу. Но если в класс Manager будет включено поле, требующее "глубокого" копирования, клонирование перестанет выполняться. Нет никакой га-

рантии, что в подклассе реализован метод clone(), корректно решающий поставленную задачу. По этой причине в классе Object метод clone() объявлен как protected. Однако, если вы хотите, чтобы пользователи ваших классов могли вызывать clone(), подобная "роскошь" остается недоступной для вас.

Следует ли реализовывать метод clone() в своих классах? Если пользователям требуется "глубокое" копирование, то ответ, конечно же, должен быть положительным. Некоторые специалисты считают, что от метода clone() надо вовсе отказаться и реализовать вместо него другой метод, выполняющий аналогичную задачу. Мы согласны, что метод clone() — не совсем удачное решение, но если вы передадите его функции другому методу, вы столкнетесь с теми же проблемами. Как бы то ни было, клонирование поддерживается не очень часто. Достаточно сказать, что метод clone() реализован менее чем в 5% классов из стандартной библиотеки.

Программа, код которой приведен в листинге 6.3, клонирует объекты класса Employee, затем вызывает два модифицирующих метода. Метод raiseSalary() изменяет значение поля salary, а метод setHireDay() — состояние поля hireDay. Ни один из модифицирующих методов не действует на исходный объект, поскольку метод clone() переопределен и выполняет "глубокое" копирование.

В главе 12 вы познакомитесь с механизмом сериализации объектов, который представляет собой альтернативу клонированию. Средства сериализации относительно легко реализовать, они не представляют опасности, но недостаточно эффективны.

Листинг 6.2. Содержимое файла CloneTest.java

```
import java.util.*;

public class CloneTest
{
   public static void main(String[] args)
   {
      try
      {
         Employee original =
            new Employee("John Q. Public", 50000);
         original.setHireDay(2000, 1, 1);
         Employee copy = original.clone();
         copy.raiseSalary(10);
         copy.setHireDay(2002, 12, 31);
         System.out.println("original=" + original);
         System.out.println("copy=" + copy);
      }
      catch (CloneNotSupportedException e)
      {
         e.printStackTrace();
      }
   }
}

class Employee implements Cloneable
{
   public Employee(String n, double s)
   {
```

```
      name = n;
      salary = s;
   }

   public Employee clone() throws CloneNotSupportedException
   {
      // Вызов Object.clone().
      Employee cloned = (Employee)super.clone();

      // Клонирование изменяемых полей.
      cloned.hireDay = (Date)hireDay.clone();

      return cloned;
   }

   /**
      Установка даты найма на работу.
      @param year год найма
      @param month месяц найма
      @param day день найма
   */
   public void setHireDay(int year, int month, int day)
   {
      hireDay = new GregorianCalendar(year,
         month - 1, day).getTime();
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public String toString()
   {
      return "Employee[name=" + name
         + ",salary=" + salary
         + ",hireDay=" + hireDay
         + "]";
   }

   private String name;
   private double salary;
   private Date hireDay;
}
```

Интерфейсы и обратный вызов

Механизм *обратного вызова* (callback) широко распространен в программировании. При обратном вызове программист задает действия, которые должны выполняться *всякий раз, когда происходит некоторое событие. Например, можно задать действие*, которое должно быть выполнено после щелчка на некоторой кнопке или при выборе определенного пункта меню. Однако, поскольку вы еще не умеете создавать пользовательский интерфейс, мы рассмотрим похожую, но более простую ситуацию.

Пакет `javax.swing` содержит класс `Timer`, который можно использовать для отсчета интервалов времени. Например, если в программе предусмотрены часы, то с помощью класса `Timer` можно отсчитывать каждую секунду и обновлять циферблат часов.

Устанавливая таймер, мы задаем интервал времени и указываем, что должно произойти по его истечении.

Как указать таймеру, что он должен делать? Во многих языках программирования задается имя функции, которую таймер должен периодически вызывать. Однако классы из стандартной библиотеки языка Java используют объектно-ориентированный подход. Программист должен передать таймеру объект некоторого класса. После этого таймер вызывает один из методов данного объекта. Передача объекта – более гибкий механизм, чем вызов функции, поскольку объект может нести дополнительную информацию.

Разумеется, таймер должен знать, какой метод он должен вызвать. Для этого таймеру нужно указать объект класса, реализующего интерфейс `ActionListener` из пакета `java.awt.event`. Этот интерфейс выглядит следующим образом:

```
public interface ActionListener
{
   void actionPerformed(ActionEvent event);
}
```

По истечении заданного интервала времени таймер вызывает метод `actionPerformed()`.

Как было сказано в главе 5, в языке Java есть аналоги указателей на функции — объекты класса `Methods`. Однако их трудно использовать, и они медленнее работают. Кроме того, на этапе компиляции не проверяются типы. Каждый раз, когда вам кажется, что указатель на функцию из языка C++ был бы уместен, рассмотрите возможность применения Java-интерфейса.

Допустим, нам нужно каждые десять секунд выводить на экран сообщение о текущем времени, сопровождаемое звуковым сигналом. Для этого необходимо определить класс, реализующий интерфейс `ActionListener`, а затем поместить операторы, которые нужно выполнить, в тело метода `actionPerformed()`.

```
class TimePrinter implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      Date now = new Date();
      System.out.println("At the tone, the time is " + now);
      Toolkit.getDefaultToolkit().beep();
   }
}
```

Обратите внимание на параметр `ActionEvent` метода `actionPerformed()`. Он содержит информацию о событии, например, об объекте, его породившем (более подробно этот вопрос рассматривается в главе 8). В данной программе детальная информация о событии не важна, поэтому можно просто проигнорировать этот параметр.

Затем следует создать объект данного класса и передать его конструктору класса Timer.

```
ActionListener listener = new TimePrinter();
Timer t = new Timer(10000, listener);
```

Первый параметр конструктора Timer представляет собой интервал времени между точками отсчета, измеренный в миллисекундах. Сообщение должно выдаваться на экран каждые десять секунд. Второй параметр является объектом класса ActionListener.

Итак, запустим таймер.

```
t.start();
```

Каждые десять секунд на экране будет появляться следующее сообщение, сопровождаемое звуковым сигналом:

```
At the tone, the time is Thu Apr 13 23:29:08 PDT 2000
```

В листинге 6.3 приведен код программы, реализующей описанный алгоритм. После запуска таймера программа выводит на экран окно с сообщением и ждет, пока пользователь не щелкнет на кнопке OK, чтобы завершить работу; при этом на экран каждые десять секунд выводится текущее время.

Будьте терпеливы. После того как диалоговое окно с вопросом Quit program? отобразится на экране, первое сообщение таймера появится только через десять секунд.

Заметим, что в дополнение к пакетам javax.swing.* и java.util.* программа импортирует класс javax.swing.Timer. Как видите, при импортировании указано имя класса. Это позволяет избежать неоднозначности. Дело в том, что существует класс java.util.Timer, который к нашей задаче отношения не имеет.

Листинг 6.3. Содержимое файла TimerTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;
// Таким образом мы разрешаем конфликт с классом java.util.Timer

public class TimerTest
{
   public static void main(String[] args)
   {
      ActionListener listener = new TimePrinter();

      // Создание таймера, вызывающего обработчик
      // каждые 10 секунд.
      Timer t = new Timer(10000, listener);
      t.start();

      JOptionPane.showMessageDialog(null, "Quit program?");
      System.exit(0);
   }
}
```

```
class TimePrinter implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      Date now = new Date();
      System.out.println("At the tone, the time is " + now);
      Toolkit.getDefaultToolkit().beep();
   }
}
```

- `static void showMessageDialog(Component parent, Object message)`

  Отображает на экране диалоговое окно со строкой сообщения и кнопкой OK. Диалоговое окно выравнивается по центру компонента `parent`. Если ссылка `parent` является нулевой, окно выравнивается по центру экрана.

javax.swing.Timer 1.2

- `Timer(int interval, ActionListener listener)`

  Создает таймер, передающий сообщение объекту `listener`. Сообщение передается по истечении интервала времени, продолжительность которого задается параметром `interval`.

- `void start()`

  Запускает таймер, который затем вызывает метод `actionPerformed()`, и передает ему объект `listener`.

- `void stop()`

  Останавливает таймер. После остановки таймера метод `actionPerformed()` больше не вызывается.

java.awt.Toolkit 1.0

- `static Toolkit getDefaultToolkit()`

  Возвращает инструментальный набор, используемый по умолчанию. Этот набор содержит информацию о среде пользовательского графического интерфейса.

- `void beep()`

  Воспроизводит звуковой сигнал.

Внутренние классы

Внутренним (inner) называется класс, определенный внутри другого класса. Зачем он нужен? Назовем три причины.

- Объект внутреннего класса имеет доступ к данным объекта, в котором он определен, включая закрытые данные.

- Внутренний класс можно скрыть от других классов того же пакета.
- *Безымянный, или анонимный* (anonimous) внутренний класс удобен, если нужно определить обратный вызов в процессе выполнения программы; при этом не приходится создавать большой объем дополнительного кода.

В данном разделе мы рассмотрим внутренние классы в таком порядке.

- Вначале обсудим простой внутренний класс, способный обращаться к полям экземпляра внешнего класса.
- Далее рассмотрим специальные синтаксические правила, используемые при объявлении внутренних классов.
- Потом речь пойдет о преобразовании внутренних классов в обычные. Начинающие программисты могут пропустить этот материал.
- После этого рассматриваются локальные внутренние классы, способные обращаться к локальным переменным в области видимости включающего класса.
- Затем мы введем понятие *безымянного внутреннего класса* (anonymous inner classes) и покажем преимущества использования таких классов при организации обратных вызовов.
- Наконец, вы увидите, как можно использовать *статические внутренние классы* (static inner classes) для формирования вложенных вспомогательных классов.

В языке C++ есть *вложенные классы* (nested classes). Вложенный класс находится в области видимости охватывающего его класса. Вот типичный пример: внутри класса для связного списка определен класс, содержащий связи, и класс, в котором определяется позиция итератора.

```
class LinkedList
{
public:
   class Iterator // Вложенный класс.
   {
   public:
       void insert(int x);
       inr erase();
       ...
   };
   ...
private:
   class Link // Вложенный класс.
   {
   public:
      Link* next;
      int data;
   };
   ...
}
```

Вложение представляет собой отношение между *классами*, а не между *объектами*. Объект класса `LinkedList` не содержит подобъектов типа `Iterator` или `Link`.

У вложения классов есть два преимущества: возможность *управления* именами (name control) и *управления доступом* (access control). Поскольку имя `Iterator` вложено внутрь класса `LinkedList`, оно известно исключительно как `LinkedList::Iterator`

и не может конфликтовать с другими классами, имеющими имя `Iterator`. В языке Java это преимущество не играет такой роли, поскольку в нем подобное управление именами осуществляют *пакеты*. Отметим, что класс `Link` находится в закрытом разделе класса `LinkedList`. Он полностью скрыт от остальной части программы. По этой причине можно объявлять его поля открытыми. Тогда методы класса `LinkedList` получат к ним доступ (который им совершенно необходим), причем для остальных методов эти поля останутся невидимыми. В языке Java такой вид управления доступом невозможен, если не применять внутренние классы.

Однако внутренние классы в языке Java имеют еще одно преимущество, которое делает их более полезными, чем вложенные классы в языке C++. Объект внутреннего класса содержит неявную ссылку на объект внешнего класса, который создал его. С помощью этой ссылки объект внутреннего класса получает доступ ко всем полям и методам внешнего объекта. Этот механизм более подробно рассматривается далее в данной главе.

В Java дополнительную ссылку не имеют лишь статические внутренние классы. Именно они представляют собой полный аналог вложенных классов языка C++.

Использование внутренних классов для определения и изменения состояния объекта

Синтаксис, используемый для работы с внутренними классами, довольно сложен. Поэтому, чтобы продемонстрировать применение внутренних классов, мы рассмотрим простой, но несколько надуманный пример. Мы преобразуем пример `TimerTest` и сформируем класс `TalkingClock`. Для организации работы "говорящих часов" применяются два параметра: интервал между последовательными сообщениями и флаг, *позволяющий включать или отключать звуковой сигнал*.

```
class TalkingClock
{
   public TalkingClock(int interval, boolean beep) { . . . }
   public void start() { . . . }

   private int interval;
   private boolean beep;

   private class TimePrinter implements ActionListener
      // Внутренний класс
   {
      . . . .
   }
}
```

Обратите внимание, что класс `TimePrinter` теперь расположен в классе `TalkingClock`. Это не значит, что каждый экземпляр `TalkingClock` содержит поле типа `TimePrinter`. Как вы увидите, объекты `TimePrinter` создаются с методами класса `TalkingClock`.

`TimePrinter` представляет собой *закрытый внутренний класс* в составе `TalkingClock`. Такой подход выбран из соображений защиты. В результате создавать объекты `TimePrinter` могут только методы класса `TalkingClock`.

Закрытыми могут быть только внутренние классы. Обычные классы должны быть либо общедоступными, либо допускать обращение из того же пакета.

Рассмотрим более подробно класс TimePrinter. Заметьте, что метод actionPerformed(), перед тем как воспроизвести звуковой сигнал, проверяет флаг beep.

```
private class TimePrinter implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      Date now = new Date();
      System.out.println("At the tone, the time is " + now);
      if (beep) Toolkit.getDefaultToolkit().beep();
   }
}
```

Теперь начинается самое интересное. Нетрудно заметить, что в составе класса TimePrinter нет поля beep. Вместо этого метод actionPerformed() обращается к соответствующему полю объекта TalkingClock. С подобным явлением мы еще не встречались. Обычно метод обращается к полям объекта. Однако внутренний класс имеет доступ не только к своим полям, но и к полям создавшего его объекта, т.е. экземпляра внешнего класса.

Для того чтобы это стало возможным, внутренний класс должен содержать ссылку на объект внешнего класса. Такая ссылка условно показана на рис. 6.3.

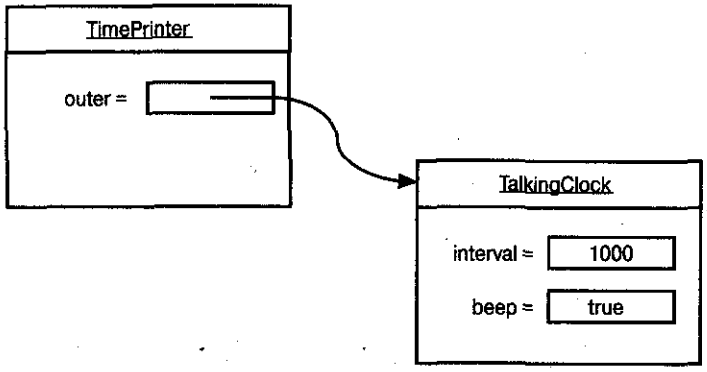


Рис. 6.3. Объект внутреннего класса содержит ссылку на объект внешнего класса

В определении внутреннего класса эта ссылка явным образом не присутствует. Для того чтобы продемонстрировать, как она работает, включим в код ссылку outer. Тогда метод actionPerformed() будет выглядеть следующим образом:

```
public void actionPerformed(ActionEvent event)
{
   Date now = new Date();
   System.out.println("At the tone, the time is " + now);
   if (outer.beep) Toolkit.getDefaultToolkit().beep();
}
```

Ссылка на объект внешнего класса задается в конструкторе. Поскольку в TalkingClock конструкторы не определены, компилятор формирует конструктор, генерируя код, подобный следующему:

```
// Автоматически генерируемый код
public TimePrinter(TalkingClock clock)
{
    outer = clock;
}
```

Обратите, пожалуйста, внимание на то, что слово `outer` не является ключевым. Оно используется для иллюстрации механизма, задействованного при работе с внутренними классами.

Если во внутреннем классе определены конструкторы, компилятор модифицирует их, добавив параметр, выполняющий роль ссылки на внешний класс.

После того как метод `start()` создаст объект класса `TimePrinter`, компилятор передаст конструктору текущего объекта ссылку `this` на объект `TalkingClock`.

```
listener = new TimePrinter(this); // Параметр добавляется автоматически
```

В листинге 6.4 показана законченная программа, иллюстрирующая работу с внутренними классами. Если бы `TimePrinter` был обычным классом, он должен был бы получить доступ к флагу `beep` посредством общедоступного метода. Наличие внутреннего класса упрощает задачу. Теперь создавать специальный метод, возвращающий значение `beep`, не обязательно.

Листинг 6.4. Содержимое файла `InnerClassTest.java`

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;

public class InnerClassTest
{
   public static void main(String[] args)
   {
      TalkingClock clock = new TalkingClock(1000, true);
      clock.start();

      // Программа выполняется до тех пор, пока
      // пользователь не щелкнет на кнопке OK
      JOptionPane.showMessageDialog(null, "Quit program?");
      System.exit(0);
   }
}

/**
   Часы, выводящие информацию о времени через
   заданный интервал.
*/
class TalkingClock
{
   /**
      Поддерживает "говорящие часы"
```

```
      @param interval Интервал в миллисекундах между сообщениями
      @param beep Значение true включает звуковой сигнал
   */
   public TalkingClock(int interval, boolean beep)
   {
      this.interval = interval;
      this.beep = beep;
   }

   /**
      Запуск часов.
   */
   public void start()
   {
      ActionListener listener = new TimePrinter();
      Timer t = new Timer(interval, listener);
      t.start();
   }

   private int interval;
   private boolean beep;

   private class TimePrinter implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         Date now = new Date();
         System.out.println("At the tone, the time is " + now);
         if (beep) Toolkit.getDefaultToolkit().beep();
      }
   }
}
```

Специальные синтаксические правила для работы с внутренними классами

В предыдущем разделе мы называли ссылку на внешний класс именем `outer`. На самом деле синтаксис, связанный с использованием внешней ссылки, несколько сложнее. Выражение

Внешний_класс`.this`

означает ссылку на внешний класс. Например, во внутреннем классе `TimePrinter` можно создать метод `actionPerformed()`.

```
public void actionPerformed(ActionEvent event)
{
   . . .
   if (TalkingClock.this.beep) Toolkit.getDefaultToolkit().beep();
}
```

И, наоборот, можно более явно записать конструктор внутреннего класса, используя следующий синтаксис:

объект_Внешнего_Класса`.new` *Внутренний_Класс*`(`*параметры*`)`

Например:

```
ActionListener listener = this.new TimePrinter();
```

Теперь ссылке на внешний класс, содержащейся во вновь созданном объекте класса TimePrinter, присвоена ссылка this метода, создавшего объект внутреннего класса. Этот случай является наиболее распространенным. Как всегда, ключевое слово this излишне. Однако ссылке на объект внешнего класса можно присвоить другой объект, явно указав его. Например, если класс TimePrinter является общедоступным внутренним классом, можно создать его объекты для любых "говорящих часов".

```
TalkingClock jabberer = new TalkingClock(1000, true);
TalkingClock.TimePrinter listener = jabberer.new TimePrinter();
```

Отметим, что если приведенная ниже ссылка находится в области видимости внешнего класса, то она относится к внутреннему классу.

```
Внешний_Класс.Внутренний_Класс
```

Например, если класс TimePrinter является общедоступным, на него можно ссылаться в любом месте программы с помощью выражения TalkingClock.TimePrinter.

Чем полезны внутренние классы

Внутренние классы были впервые реализованы в JDK 1.1. Многие программисты встретили появление внутренних классов настороженно, считая их отступлением от принципов, заложенных в основу языка Java. По их мнению преимуществом языка Java перед C++ является его простота. Внутренние классы действительно сложны. (Такой вывод вы наверняка сделаете, познакомившись с безымянными внутренними классами, которые будут рассматриваться далее в этой главе.) Взаимодействие внутренних классов с другими средствами языка не совсем очевидно, в особенности это касается управления доступом.

Зачем же разработчики языка Java пожертвовали преимуществами, которыми он выгодно отличался от других языков, в пользу элегантного и интересного механизма, выгода от которого настолько неочевидна?

У нас нет окончательного ответа на этот вопрос. Стоит уточнить, что обработку внутренних классов осуществляет *компилятор*, а не виртуальная машина. Для их обозначения используется символ $, разделяющий имена внешних и внутренних классов; таким образом, для виртуальной машины внутренние классы неотличимы от внешних.

Например, класс TimePrinter в составе TalkingClock преобразуется в класс TalkingClock$TimePrinter. Чтобы увидеть, как работает этот механизм, попробуйте провести следующий эксперимент: запустите программу ReflectionTest (см. главу 5) и выполните отражение класса TalkingClock$TimePrinter. На экране появятся следующие строки:

```
class TalkingClock$TimePrinter
{
   private TalkingClock$TimePrinter(TalkingClock);
   TalkingClock$TimePrinter(TalkingClock, TalkingClock$1);

   public void actionPerformed(java.awt.event.ActionEvent);

   final TalkingClock this$0;
}
```

Программируя в среде Unix, не используйте символ $, если имя класса указывается в командной строке. Это значит, что программу ReflectionTest следует запускать с помощью команды

```
java ReflectionTest  'TalkingClock$TimePrinter'
```

Нетрудно заметить, что компилятор генерирует дополнительное поле this$0 для ссылки на внешний класс. (Имя this$0 синтезируется компилятором, поэтому сослаться на него невозможно.) Количество параметров конструктора также увеличивается. В действиях конструктора разобраться не просто. Закрытый конструктор устанавливает поле this$0. Конструктор, область видимости которого ограничивается пакетом, содержит второй параметр типа TalkingClock$1 (класс, доступный из пакета, не содержащий полей и методов). Экземпляр этого класса никогда не создается. Класс TalkingClock содержит следующий вызов:

```
new TalkingClock$TimePrinter(this, null)
```

Если компилятор автоматически выполняет преобразования, нельзя ли реализовать подобный механизм вручную? Попробуем сделать это. Превратим TimePrinter в обычный класс, определяемый за пределами TalkingClock, а затем передадим ему ссылку this на создавший его объект.

```
class TalkingClock
{
   . . .
   public void start()
   {
      ActionListener listener = new TimePrinter(this);
      Timer t = new Timer(interval, listener);
      t.start();
   }
}

class TimePrinter implements ActionListener
{
   public TimePrinter(TalkingClock clock)
   {
         outer = clock;
   }
   . . .
   private TalkingClock outer;
}
```

Рассмотрим теперь метод actionPerformed(). Ему нужен доступ к полю outer.beep.

```
if (outer.beep) . . . // Ошибка!
```

Здесь возникает проблема. Внутренний класс имеет доступ к закрытым данным лишь внешнего класса, который его содержит, но теперь класс TimePrinter не является внутренним.

Следовательно, внутренние классы намного мощнее, чем обычные, поскольку обладают более высокими привилегиями доступа.

Может возникнуть вопрос, каким образом внутренние классы получают дополнительные привилегии доступа, ведь они преобразуются в обычные, а виртуальной машине вообще о них ничего не известно? Чтобы раскрыть эту тайну, снова воспользуемся программой `ReflectionTest`, отслеживающей поведение класса `TalkingClock`.

```
class TalkingClock
{
   public TalkingClock(int, boolean);

   static boolean access$100(TalkingClock);
   public void start();

   private int interval;
   private boolean beep;
}
```

Обратите внимание на статический метод `access$100`, добавленный компилятором во внешний класс. Он возвращает значение поля `beep`, содержащегося в объекте, который передан методу в качестве параметра.

Внутренний класс вызывает этот метод. В методе `actionPerformed()` класса `TimePrinter` есть следующее выражение:

```
if (beep)
```

Оно преобразуется в приведенный ниже вызов.

```
if (access$100(outer));
```

Не опасно ли это? В принципе, да! Посторонний может легко вызвать метод `access$100()` и прочесть данные закрытого поля `beep`. Конечно, `access$100` не является допустимым именем для Java-метода. Однако хакеры, знакомые со структурой файлов, содержащих классы, легко могут создать свой аналогичный файл и вызвать данный метод с помощью соответствующих команд виртуальной машины. Разумеется, такой файл должен формироваться вручную (например, с помощью редактора шестнадцатеричных кодов). Но поскольку область видимости секретных методов доступа ограничена пакетом, атакующий код должен размещаться в том же пакете, что и атакуемый класс.

Итак, если внутренний класс обладает доступом к закрытым полям, можно создать другой класс, добавить его в тот же пакет и получить доступ к закрытым данным. Правда, для этого нужен опыт и решительность. Программист не может получить такой доступ случайно, не создавая для этих целей специальный файл, содержащий модифицированные классы.

Локальные внутренние классы

Если внимательно изучить код класса `TalkingClock`, обнаружится, что имя класса `TimePrinter` используется лишь однажды — при создании объекта этого типа в методе `start()`.

В таких ситуациях класс можно определить *локально в отдельном методе.*

```
public void start()
{
   class TimePrinter implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         Date now = new Date();
         System.out.println("At the tone, the time is " + now);
         if (beep) Toolkit.getDefaultToolkit().beep();
      }
   }

   ActionListener listener = new TimePrinter();
   Timer t = new Timer(1000, listener);
   t.start();
}
```

Локальные классы никогда не объявляются с помощью модификаторов доступа (т.е. public и protected). Их область видимости всегда ограничивается блоком, в котором они объявлены.

Локальные классы имеют огромное преимущество — они полностью скрыты от внешнего мира и даже от остальной части класса TalkingClock. Ни один метод, за исключением метода start(), ничего не знает о классе TimePrinter.

Кроме того, локальные классы выгодно отличаются от обычных внутренних тем, что они имеют доступ не только к полям своего внешнего класса, но и к локальным переменным! Рассмотрим типичный пример. Переместим параметры interval и beep из конструктора TalkingClock в метод start().

```
public void start(int interval, final boolean beep)
{
   class TimePrinter implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         Date now = new Date();
         System.out.println("At the tone, the time is " + now);
         if (beep) Toolkit.getDefaultToolkit().beep();
      }
   }

   ActionListener listener = new TimePrinter();
   Timer t = new Timer(1000, listener);
   t.start();
}
```

Обратите внимание на то, что классу TimePrinter больше не нужно хранить переменную экземпляра beep. Он просто ссылается на параметр метода, содержащего определение данного класса.

Возможно, это не так уж и неожиданно. В конце концов, приведенная ниже строка находится внутри метода start(), так почему бы ей не иметь доступа к переменной beep?

```
if (beep) . . .
```

Локальные внутренние классы обладают рядом особенностей. Чтобы понять их, рассмотрим поток управления.

1. Вызывается метод `start()`.
2. С помощью вызова конструктора внутреннего класса `TimePrinter` инициализируется объектная переменная `listener`.
3. Ссылка `listener` передается конструктору класса `Timer`, таймер запускается, и метод `start()` прекращает свою работу. В этот момент параметр `beep` метода `start()` больше не существует.
4. Впоследствии метод `actionPerformed()` выполняет оператор `if (beep) . . .;`

Чтобы метод `actionPerformed()` выполнялся успешно, класс `TimePrinter` должен создать копию поля `beep` до того, как оно перестанет существовать в качестве локальной переменной метода `start()`. Именно это и происходит. В нашем примере компилятор синтезирует для локального внутреннего класса имя `TalkingClock$1TimePrinter`. Применив снова программу `ReflectionTest` для изучения класса `TalkingClock$1TimePrinter`, получим следующий результат:

```
class TalkingClock$1TimePrinter
{
   TalkingClock$1TimePrinter(TalkingClock, boolean);

   public void actionPerformed(java.awt.event.ActionEvent);

   final boolean val$beep;
   final TalkingClock this$0;
}
```

Обратите внимание на параметр конструктора, имеющий тип `boolean`, и переменную экземпляра `val$beep`. При создании объекта переменная `beep` передается конструктору и размещается в поле `val$beep`. Для того чтобы это стало возможным, разработчикам компилятора пришлось достаточно потрудиться. Компилятор должен обнаруживать доступ к локальным переменным, создавать для каждой из них соответствующие поля, а затем копировать локальные переменные в конструкторе так, чтобы поля данных инициализировались копиями локальных переменных.

С точки зрения разработчика прикладных программ, доступ к локальным переменным выглядит привлекательно. Благодаря такой возможности внутренние классы становятся проще и уменьшается количество полей, которые должны программироваться явным образом.

Как уже упоминалось, методы локального класса могут ссылаться только на локальные переменные, объявленные с помощью ключевого слова `final`. По этой причине параметр `beep` в нашем примере был объявлен терминальным. Терминальная локальная переменная не может быть модифицирована. Этим гарантируется, что локальная переменная и ее копия, созданная внутри локального класса, всегда имеют одно и то же значение.

Мы уже видели, что терминальные переменные можно использовать в качестве констант:

```
public static final double SPEED_LIMIT = 55;
```

Ключевое слово `final` можно применять и к локальным, статическим переменным, а также к переменным экземпляра. Во всех случаях это означает одно и то же: данной переменной значение присваивается только один раз, сразу после ее создания. Впоследствии это значение изменить будет невозможно.

Однако инициализировать терминальную переменную одновременно с ее определением не обязательно. Например, параметр `beep` инициализируется после создания при вызове метода `start`. (Если этот метод вызывается неоднократно, при каждом вызове будет создаваться новый параметр `beep`.) Значение переменной экземпляра `val$beep` в классе `TalkingClock$1TimePrinter` устанавливается лишь один раз, в конструкторе внутреннего класса. Терминальную переменную, которая не была инициализирована при определении, часто называют *пустой терминальной переменной* (blank final variable).

Безымянные внутренние классы

Работая с локальными внутренними классами, можно использовать еще одну интересную возможность. Если нужно создать один-единственный объект данного класса, этому классу можно даже не присваивать имени. Такой класс называется *безымянным, или анонимным внутренним классом* (anonimous inner class).

```
public void start(int interval, final boolean beep)
{
   ActionListener listener = new
      ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
             Date now = new Date();
             System.out.println(
                "At the tone, the time is " + now);
             if (beep) Toolkit.getDefaultToolkit().beep();
         }
      };
   Timer t = new Timer(1000, listener);
   t.start();
}
```

Надо признать, что синтаксис достаточно сложен. На самом деле приведенный выше фрагмент означает следующее: создается новый объект класса, реализующего интерфейс `ActionListener`, в котором в фигурных скобках {} определен требуемый метод `actionPerformed()`.

Любые параметры, необходимые для создания объекта, задаются в круглых скобках, следующих за именем супертипа. В общем случае синтаксис выглядит следующим образом:

```
new СуперТип(параметры, необходимые для создания объекта)
{
  Методы внутреннего класса и данные.
}
```

В данном случае *СуперТип* — это интерфейс `ActionListener`; внутренний класс *реализует* данный интерфейс. *СуперТип* может быть и классом; тогда внутренний класс *расширяет* данный суперкласс.

Безымянный внутренний класс не может иметь конструкторов, поскольку имя конструктора должно совпадать с именем класса, а в данном случае класс не имеет имени. Вместо этого параметры, необходимые для создания объекта, передаются конструктору *суперкласса*. Казалось бы, если внутренний класс реализует какой-либо интерфейс, параметры не задаются, однако существует способ указать их.

Следует тщательно изучить различие между созданием нового объекта некоторого класса и созданием объекта безымянного внутреннего класса, расширяющего данный класс. Если вслед за скобками, содержащими список параметров, необходимых для создания объекта, указана открытая фигурная скобка, определяется безымянный внутренний класс.

```
Person queen = new Person("Mary");
   // Объект класса Person.
Person count = new Person("Dracula") {...};
   // Объект внутреннего класса, расширяющего класс Person.
```

Идея ввести в язык безымянные внутренние классы оказалась удачной, или это только великолепное средство создавать никому не понятные программы? Увы, и то и другое одновременно. Если внутренний класс невелик и сводится всего к нескольким строчкам простого кода, можно сберечь время на указании типа, но сэкономленное время обернется трудностями при чтении исходного текста программы.

Досадно, что разработчики не предприняли никаких попыток улучшить синтаксис безымянных внутренних классов, так как в остальном синтаксис языка Java намного проще, чем синтаксис языка C++. Разработчики внутренних классов могли бы помочь программистам разобраться с такими синтаксическими конструкциями, как, например, следующая:

```
Person count = new class extends Person("Dracula") {...};
   // Синтаксис, не свойственный языку Java.
```

Однако они этого не сделали. Поскольку многие программисты считают, что коды, содержащие безымянные внутренние классы, слишком сложны для понимания, мы рекомендуем ограничивать их применение.

В листинге 6.5 приведена законченная программа, реализующая "говорящие часы", в которой используется безымянный внутренний класс. Сравнив эту программу с кодом, содержащимся в листинге 6.4, вы можете убедиться, что применение безымянного внутреннего класса сделало программу несколько короче, но не проще для понимания.

Листинг 6.5. Содержимое файла AnonymousInnerClassTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;

public class AnonymousInnerClassTest
{
   public static void main(String[] args)
   {
      TalkingClock clock = new TalkingClock();
      clock.start(1000, true);

      // Программа выполняется до тех пор, пока
```

```
      // пользователь не щелкнет на кнопке OK
      JOptionPane.showMessageDialog(null, "Quit program?");
      System.exit(0);
   }
}

/**
   Часы, выводящие показания времени
   через определенный интервал.
*/
class TalkingClock
{
   /**
      Запуск часов.
      @param interval Интервал времени в миллисекундах
         между сообщениями
      @param beep Значение true включает звуковой сигнал
   */
   public void start(int interval, final boolean beep)
   {
      ActionListener listener = new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               Date now = new Date();
               System.out.println(
                  "At the tone, the time is " + now);
               if (beep) Toolkit.getDefaultToolkit().beep();
            }
         };
      Timer t = new Timer(interval, listener);
      t.start();
   }
}
```

Статические внутренние классы

Может случиться так, что вам нужно будет использовать внутренний класс просто для того, чтобы скрыть его внутри другого класса, а ссылка на объект внешнего класса окажется ненужной. Подавить генерацию этой ссылки можно, объявив внутренний класс статическим (`static`).

Предположим, нам нужно найти в массиве максимальное и минимальное число. Разумеется, для этого мы напишем два метода: один — для нахождения максимального, а другой — для минимального числа. При вызове обеих функций массив просматривается дважды. Было бы намного эффективнее просматривать массив только один раз, одновременно определяя и максимальное, и минимальное число.

```
double min = Double.MAX_VALUE;
double max = Double.MIN_VALUE;
for (double v : values)
{
   if (min > v) min = v;
   if (max < v) max = v;
}
```

Однако в таком случае функция должна возвращать два значения. Сделать это можно, определив класс Pair, содержащий два числа.

```
class Pair
{
   public Pair(double f, double s)
   {
      first = f;
      second = s;
   }
   public double getFirst() { return first; }
   public double getSecond() {  return second; }

   private double first;
   private double second;
}
```

Тогда метод minmax сможет вернуть объект класса Pair.

```
class ArrayAlg
{
   public static Pair minmax(double[] values)
   {
      . . .
      return new Pair(min, max);
   }
}
```

Вызывающий метод использует для получения минимального и максимального значения методы getFirst() и getSecond().

```
Pair p = ArrayAlg.minmax(d);
System.out.println("min = " + p.getFirst());
System.out.println("min = " + p.getSecond());
```

Разумеется, имя Pair слишком неконкретно, и при выполнении большого проекта у другого программиста может возникнуть такая же блестящая идея, вот только класс Pair у него будет содержать не числа, а строки. Эту потенциальную проблему можно решить, сделав класс Pair внутренним классом, определенным внутри ArrayAlg. Тогда настоящим именем класса будет не Pair, а ArrayAlg.Pair.

```
ArrayAlg.Pair p = ArrayAlg.minmax(d);
```

Однако в отличие от внутренних классов, использованных в предыдущих примерах, ссылка на другой объект внутри класса Pair не нужна. Ее можно подавить, объявив внутренний класс статическим.

```
class ArrayAlg
{
   public static class Pair
   {
      ...
   }
      ...
}
```

Разумеется, только внутренние классы можно объявлять статическими. Статический внутренний класс ничем не отличается от любого другого внутреннего класса, за исключением того, что его объект не содержит ссылку на создавший его объект внешнего класса. В нашем примере следует применять статический внутренний класс, поскольку объект внутреннего класса создается внутри статического метода.

```
public static Pair minmax(double[] d)
{
   ...
   return new Pair(min, max);
}
```

Если бы класс `Pair` не был объявлен статическим, компилятор сообщил бы о том, что при инициализации объекта внутреннего класса объект типа `ArrayAlg` не доступен.

Статический внутренний класс используется тогда, когда доступ к объекту внешнего класса не нужен. Некоторые программисты для обозначения статических внутренних классов используют термин *вложенные классы* (nested classes).

Внутренние классы, определенные в интерфейсах автоматически, считаются статическими и общедоступными.

В листинге 6.6 приведен полный код класса `ArrayAlg` и вложенного класса `Pair`.

Листинг 6.6. Содержимое файла `StaticInnerClassTest.java`

```
public class StaticInnerClassTest
{
   public static void main(String[] args)
   {
      double[] d = new double[20];
      for (int i = 0; i < d.length; i++)
         d[i] = 100 * Math.random();
      ArrayAlg.Pair p = ArrayAlg.minmax(d);
      System.out.println("min = " + p.getFirst());
      System.out.println("max = " + p.getSecond());
   }
}

class ArrayAlg
{
   /**
      Два числа с плавающей точкой
   */
   public static class Pair
   {
      /**
          Объединение двух чисел с плавающей точкой
          @param f Первое число
          @param s Второе число
      */
      public Pair(double f, double s)
      {
```

```
        first = f;
        second = s;
    }

    /**
       Возвращает первое число из пары.
       @return Первое число
    */
    public double getFirst()
    {
       return first;
    }

    /**
       Возвращает второе число из пары.
       @return Второе число
    */
    public double getSecond()
    {
       return second;
    }

    private double first;
    private double second;
}

/**
   Определяет минимальное и максимальное число в массиве.
   @param d Массив чисел с плавающей точкой
   @return Пара, в которой первый элемент
     является минимальным числом массива,
     а второй элемент — максимальным числом.
*/
public static Pair minmax(double[] values)
{
   double min = Double.MAX_VALUE;
   double max = Double.MIN_VALUE;
   for (double v : values)
   {
      if (min > v) min = v;
      if (max < v) max = v;
   }
   return new Pair(min, max);
}
}
```

Proxy-классы

В последнем разделе этой главы мы обсудим понятие proxy-классов, которые также часто называют классами-посредниками или заместителями. Этот новый механизм стал доступным в версии 1.3 пакета JDK. Он используется для того, чтобы во время выполнения программы создавать новые классы, реализующие заданные интерфейсы. Proxy-классы необходимы, если на этапе компиляции программист еще не знает, какие интерфейсы ему следует реализовать. В прикладном программировании такая ситуация возникает крайне редко. Однако в некоторых приложениях системного программиро-

вания гибкость, обеспечиваемая proxy-классами, может оказаться очень важным обстоятельством. Используя эту концепцию, часто удается избежать механической генерации и компиляции кода заглушек.

Классы-заглушки используются в ряде случаев при создании программ. Например, если вы применяете средства Remote Method Invocation (RMI), заглушки генерируются специальной утилитой под названием `rmic` и включаются в программу. (Подробно механизм RMI будет рассматриваться в главе 5 второго тома.) Заглушки также создаются и компилируются при соединении компонентов bean. (Java beans будут обсуждаться в главе 8 второго тома.) С версии JDK 5.0, средства proxy позволяют генерировать заглушки RMI, не прибегая к помощи специальной утилиты.

Допустим, у нас есть массив объектов `Class`, представляющих интерфейсы (возможно, только один интерфейс), точная природа которых во время компиляции нам не известна. Теперь предположим, что нам нужно создать объект класса, реализующего эти интерфейсы. Это трудная проблема. Если объект класса `Class` представляет настоящий класс, мы можем применить метод `newInstance()` или задействовать механизм отражения, чтобы найти конструктор этого класса. Однако создать объект интерфейса невозможно. Следовательно, определить новый класс во время выполнения программы мы не можем.

Чтобы разрешить эту проблему, некоторые программы, например BeanBox в ранних версиях пакета Bean Development Kit, генерируют код, помещают его в файл, вызывают компилятор, а затем полученный файл класса. Естественно, это очень медленный процесс, а кроме того, при выполнении такой программы постоянно нужен компилятор. Механизм proxy предоставляет более удачное решение. Proxy-класс может создавать во время выполнения программы совершенно новые классы и реализует интерфейсы, которые указывает программист. В частности, в proxy-классе содержатся следующие методы.

- Все методы, которые требуют указанные интерфейсы.
- Все методы, определенные в классе `Object` (`toString()`, `equals()` и т.д.).

Однако определить новый код для этих методов в ходе выполнения программы нельзя. Вместо этого программист должен предоставить *обработчик вызовов* (invocation handler), представляющий собой объект, реализующий интерфейс `InvocationHandler`. В этом интерфейсе объявлен единственный метод:

```
Object invoke(Object proxy, Method method, Object[] args)
```

При вызове какого-либо метода из proxy-объекта автоматически вызывается метод `invoke()` обработчика вызовов, получающий на вход объект класса `Method` и параметры исходного вызова. Обработчик вызовов должен уметь обработать вызов.

Чтобы создать proxy-объект, используется метод `newProxyInstance()` класса `Proxy`. Этот метод получает три параметра.

- *Загрузчик класса* (class loader). Модель безопасности языка Java позволяет использовать разные загрузчики классов: для системных классов, для классов, полученных из Internet, и т.д. Загрузчики классов обсуждаются в томе 2. Пока мы будем указывать ссылку `null`, чтобы использовать загрузчик классов, предусмотренный по умолчанию.

- Массив объектов класса `Class` — по одному для каждого интерфейса, подлежащего реализации.
- Обработчик вызовов.

Остаются два вопроса. Как определить обработчик? Что можно сделать с полученным в результате proxy-объектом? Разумеется, ответы на эти вопросы зависят от задачи, которую программист хочет решить с помощью механизма proxy. Их можно использовать для достижения следующих целей.

- Переадресовывать вызовы методов на удаленный сервер.
- Устанавливать связь между событиями пользовательского интерфейса и определенными действиями, выполняемыми при работе программы.
- Отслеживать вызовы методов при отладке.

В примере, который будет рассмотрен ниже, мы применим proxy-объекты и обработчики вызовов для отслеживания обращений к методам. Мы определим класс оболочки `TraceHandler`, в котором будет храниться некоторый объект. Его метод `invoke()` лишь выводит на экран имя и параметры метода, к которому произошло обращение, а затем вызывает сам метод, задавая в качестве неявного параметра объект, хранящийся в классе `TraceHandler`.

```
class TraceHandler impements InvocationHandler
{
   public TraceHandler(Object t)
   {
     target = t;
   }

   public Object invoke(Object proxy, Method m, Object[] args)
      throws Throwable
   {
      // Вывод имени метода и его параметров.
      ...
      // Вызов метода.
      return m.invoke(target, args);
   }
   private Object target;
}
```

Ниже показано, как создать proxy-объект, позволяющий отслеживать вызов метода.

```
Object value = ...;
// Создание объекта класса оболочки.
InvocationHandler handler = new TraceHandler(value);
// Создание proxy-объекта для всех интерфейсов.
Class[] interfaces = value.getClass().getInterface();
Object proxy = Proxy.newProxyInstance(null, interfaces, handler);
```

Теперь каждый раз, когда метод вызывается из объекта `proxy`, его имя и параметры выводятся, а затем происходит обращение к соответствующему методу объекта `value`.

В программе, приведенной в листинге 6.7, мы применили proxy-объекты для отслеживания процесса поиска методом дихотомии. Сначала заполняется массив,

состоящий из объектов-заместителей для целых чисел от 1 до 1000. Затем для поиска случайного целого числа в массиве вызывается метод binarySearch класса Arrays. В заключение на экран выводится соответствующий элемент.

```
Object[] elements = new Object[1000];
// Массив заполняется proxy-объектами
// для целых чисел от 1 до 1000.
for (int i = 0; i < elements.length; i++)
{
   Integer value = i + 1;
   elements[i] = . . .; // proxy для соответствующих значений;
}

// Генерация случайного числа.
Integer key = new Random().nextInt(elements.length) + 1;

// Поиск числа key.
int result = Arrays.binarySearch(elements, key);

// Вывод на экран найденного числа
// (если оно есть в массиве).
if (result >= 0) System.out.println(elements[result]);
```

Класс Integer реализует интерфейс Comparable. Proxy-объекты принадлежат классу, определяемому во время выполнения программы. (Его имя $Proxy0.) Этот класс также реализует интерфейс Comparable. Однако его метод compareTo() вызывает метод invoke из обработчика вызовов в proxy-объекте.

Как было сказано в начале данной главы, начиная с JDK 5.0 в классе Integer был реально реализован интерфейс Comparable<Integer>. Однако во время выполнения программы информация о типах удаляется и при создании proxy используется обычный объект Comparable.

Метод binarySearch() осуществляет вызов наподобие следующего:

```
if (elements[i].compareTo(key) < 0) ...
```

Поскольку массив заполнен proxy-объектами, метод compareTo() вызывает метод invoke() класса TraceHandler. Этот метод выводит имя вызванного метода и его параметры, а затем вызывает метод compareTo() объекта Integer.

В конце программы выводятся результаты ее работы.

```
System.out.println(elements[result]);
```

Метод println() вызывает метод toString() proxy-объекта, а затем это обращение также перенаправляется обработчику вызовов.

Ниже показаны результаты полной трассировки программы.

```
500.compareTo(288)
250.compareTo(288)
375.compareTo(288)
312.compareTo(288)
281.compareTo(288)
296.compareTo(288)
```

```
288.compareTo(288)
288.toString()
```

Как видите, при поиске методом дихотомии на каждом шаге интервал уменьшается вдвое.

Листинг 6.7. Содержимое файла `ProxyTest.java`

```
import java.lang.reflect.*;
import java.util.*;

public class ProxyTest
{
   public static void main(String[] args)
   {
      Object[] elements = new Object[1000];

      // Массив заполняется proxy-объектами
      // для целых чисел от 1 до 1000.
      for (int i = 0; i < elements.length; i++)
      {
         Integer value = i + 1;
         Class[] interfaces = value.getClass().getInterfaces();
         InvocationHandler handler = new TraceHandler(value);
         Object proxy = Proxy.newProxyInstance(null,
            interfaces, handler);
         elements[i] = proxy;
      }

      // Генерация случайного числа.
      Integer key = new Random().nextInt(elements.length) + 1;

      // Поиск числа key.
      int result = Arrays.binarySearch(elements, key);

      // Вывод на экран найденного числа
      // (если оно есть в массиве).
      if (result >= 0) System.out.println(elements[result]);
   }
}

/**
   Обработчик вызова выводит имя метода и параметры,
   а затем вызывает требуемый метод.
*/
class TraceHandler implements InvocationHandler
{
   /**
      Создание объекта TraceHandler
      @param t Неявный параметр при вызове метода
   */
   public TraceHandler(Object t)
   {
      target = t;
   }

   public Object invoke(Object proxy, Method m, Object[] args)
                           throws Throwable
```

```
   {
      // Вывод неявного параметра.
      System.out.print(target);
      // Вывод имени метода.
      System.out.print("." + m.getName() + "(");
      // Вывод явных параметров.
      if (args != null)
      {
         for (int i = 0; i < args.length; i++)
         {
            System.out.print(args[i]);
            if (i < args.length - 1)
               System.out.print(", ");
         }
      }
      System.out.println(")");

      // Вызов реального метода.
      return m.invoke(target, args);
   }

   private Object target;
}
```

Свойства proxy-классов

Теперь, увидев proxy-классы в действии, вернемся к анализу некоторых их свойств. Напомним, что proxy-классы создаются во время выполнения программы. Однако затем они становятся обычными классами, как и все остальные классы, обрабатываемые виртуальной машиной.

Все proxy-классы расширяют класс `Proxy`. Такой класс содержит только одну переменную экземпляра, которая ссылается на обработчик вызовов, определенный в суперклассе `Proxy`. Любые дополнительные данные, необходимые для выполнения задач, поставленных перед proxy-объектами, должны храниться в обработчике вызовов. Например, в листинге 6.7 при создании proxy-объектов, реализующих интерфейс `Comparable`, обработчик `TraceHandler` является классом оболочки для фактических объектов.

Во всех proxy-классах переопределяются методы `toString()`, `equals()` и `hashCode()` класса `Object`. Эти методы лишь вызывают метод `invoke()`, принадлежащий обработчику событий. Другие методы класса `Object` (например, `clone()` и `getClass()`) не переопределяются.

Имена proxy-классов не определены. В виртуальной машине, разработанной Sun, генерируются имена классов, начинающиеся строкой `$Proxy`.

Для конкретного загрузчика классов и заданного набора интерфейсов может существовать только один proxy-класс. Это значит, что, вызывая метод `newProxyInstance()` дважды для одного и того же загрузчика классов и массива интерфейсов, программист получает два объекта одного и того же класса. Имя этого класса можно определить с помощью метода `getProxyClass()`.

```
Class proxyClass = Proxy.getProxyClass(null, interfaces);
```

Proxy-класс всегда является общедоступным и терминальным. Если все интерфейсы, которые реализуются proxy-классом, объявлены как `public`, этот класс не принадлежит ни одному конкретному пакету. В противном случае все интерфейсы, в объявлении которых не указан модификатор `public`, а следовательно, и сам proxy-класс, должны принадлежать одному пакету.

Вызвав метод `isProxyClass()` класса `Proxy`, можно проверить, представляет ли объект `Class` определенный proxy-класс.

java.lang.reflect.InvocationHandler 1.3

- `Object invoke(Object proxy, Method method, Object[] args)`

  Этот метод определяется для того, чтобы задать действие, которое должно быть выполнено при вызове какого-либо метода proxy-объекта.

java.lang.reflect.Proxy 1.3

- `static Class getProxy(ClassLoader loader, Class[] interfaces)`

  Возвращает proxy-класс, реализующий заданные интерфейсы.

- `static Object newProxyInstance(ClassLoader loader, Class[] interfaces,`
  `InvocationHandler handler)`

  Создает новый экземпляр proxy-класса, реализующего заданные интерфейсы. Все методы вызывают метод `invoke()` обработчика вызовов.

- `static boolean isProxyClass(Class c)`

  Возвращает значение `true`, если `c` является proxy-классом.

Эта глава завершает наше описание основ языка Java. Понятия интерфейсов и внутренних классов часто используются при создании прикладных программ. В отличие от них, proxy-классы в основном представляют интерес для разработчиков инструментальных средств.

Теперь можно переходить к изучению графики и пользовательских интерфейсов, описание которых начинается с главы 7.

Глава 7

Программирование графики

В этой главе...

- Общие сведения о пакете Swing
- Создание фреймов
- Позиционирование фреймов
- Отображение информации на панели
- Двухмерные фигуры
- Работа с цветом
- Использование шрифтов при выводе текста
- *Дополнительные операции с изображениями*

До сих пор мы учились писать программы, входные данные для которых вводились с клавиатуры. Затем программа обрабатывала их и выводила результаты на экран. Сейчас такие программы уже не соответствуют запросам большинства пользователей. Современные программы работают совершенно иначе. В этой главе рассматриваются вопросы создания графического пользовательского интерфейса (Graphics User Interface – GUI). *В частности, вы узнаете, каким образом можно изменить размер* и положение окон на экране, отобразить в окне текст, набранный разными шрифтами, вывести на экран рисунок и т.д. Эти навыки окажутся весьма полезными при создании интересных программ, описанных в следующих главах.

Главы 8 и 9 посвящены обработке таких событий, как нажатие клавиши и щелчок мыши, а также средствам создания элементов пользовательского интерфейса, например меню и кнопок. Прочитав эту главу и две следующие, вы сможете создавать независимые графические приложения. В главе 10 показано, как программировать аплеты и включать их на Web-страницы. Более сложные вопросы, связанные с программированием графики, рассматриваются в томе 2.

Общие сведения о пакете Swing

Первая версия языка Java содержала библиотеку классов Abstract Window Toolkit (AWT), предоставляющую основные средства программирования графического пользовательского интерфейса. Создание элементов пользовательского интерфейса на конкретной платформе (Windows, Solaris, Macintosh и т.д.) библиотека AWT поручала встроенным инструментальным средствам. Например, если с помощью библиотеки AWT на экран нужно было вывести окно с текстом, то фактически оно отображалось с помощью платформенно-ориентированных средств. Теоретически созданные таким образом программы должны работать на любых платформах.

Подход, основанный на использовании платформенно-ориентированных компонентов, очень хорош для простых приложений. Однако вскоре стало ясно, что с его помощью крайне трудно создавать высококачественные графические библиотеки, не зависящие от платформы. Элементы пользовательского интерфейса, например меню, панели прокрутки и текстовые поля, на разных платформах могут вести себя поразному. Следовательно, на основе этого подхода трудно создавать программы с предсказуемым поведением. Кроме того, некоторые графические среды программирования (например, X11/Motif) не имеют такого богатого набора компонентов пользовательского интерфейса, как операционные системы Windows или Macintosh. Это, в свою очередь, делало библиотеки еще более зависимыми. В результате графические приложения, созданные с помощью библиотеки AWT, выглядели по сравнению с программами для Windows или Macintosh не так привлекательно и не имели таких функциональных возможностей. Что еще хуже, в библиотеке AWT на разных платформах обнаруживались *разные* ошибки. Программистам приходилось тестировать каждое приложение на каждой платформе.

В 1996 году компания Netscape создала библиотеку программ для создания графического пользовательского интерфейса и назвала ее IFC (Internet Foundation Classes). Эта библиотека была основана на совершенно других принципах. Элементы пользовательского интерфейса, такие как меню, кнопки и т.п., *изображались* в пустом окне. Платформенно-ориентированные средства лишь учитывались при выборе способа вывода окон. Таким образом, элементы графического пользовательского интерфейса, созданные с помощью библиотеки IFC, выглядели и вели себя точно так же, как и соответствующие элементы AWT, но не зависели от платформы, на которой запускалась программа. Компании Sun и Netscape объединили свои усилия и усовершенствовали этот подход, создав библиотеку под кодовым названием Swing. С тех пор слово Swing стало официальным названием набора инструментальных средств для создания *машинонезависимого пользовательского интерфейса. Средства Swing доступны* в расширении Java 1.1 и являются частью JDK 1.2.

Библиотека Swing не может полностью заменить собой библиотеку AWT. Она лишь дает пользователю больше возможностей для создания пользовательского интерфейса. Основная архитектура библиотеки, в частности обработка событий, остается той же, используете ли вы AWT или Swing. В дальнейшем под словом Swing мы будем иметь в виду машинонезависимые классы для создания пользовательского интерфейса, а под аббревиатурой AWT — платформенно-ориентированные средства для работы с окнами.

Разумеется, элементы пользовательского интерфейса библиотеки Swing появляются на экране немного медленнее, чем компоненты, предусмотренные в библиотеке AWT. Однако на современных машинах это практически незаметно. С другой стороны, у библиотеки Swing есть несколько весьма существенных преимуществ.

- Она содержит более богатый и удобный набор элементов пользовательского интерфейса.
- Эта библиотека намного меньше зависит от платформы, на которой будет выполняться программа. Следовательно, она содержит меньше ошибок, характерных для конкретной платформы.
- Библиотека Swing обеспечивает пользователям однотипные средства для работы на разных платформах.

Все это значит, что библиотека Swing в конце концов позволит выполнить обещание компании Sun: "Написано однажды, работает везде!".

Третье достоинство библиотеки Swing может обернуться недостатком: если элементы пользовательского интерфейса на разных платформах выглядят одинаково, то (по крайней мере на некоторых платформах) они обязательно будут *отличаться* от компонентов, реализуемых в данной системе, т.е. пользователи будут в них хуже разбираться.

В библиотеке Swing эта проблема решена очень элегантно. Программисты, использующие библиотеку Swing, могут придать своей программе внешний вид в заданном стиле. Например, на рис. 7.1 и 7.2 показано, как выглядит одна и та же программа в средах Windows и Motif. (По очевидным причинам, связанным с авторскими правами, средства, позволяющие адаптировать программу для платформ Windows и Macintosh, доступны лишь в исполняемых системах, ориентированных на эти платформы.)

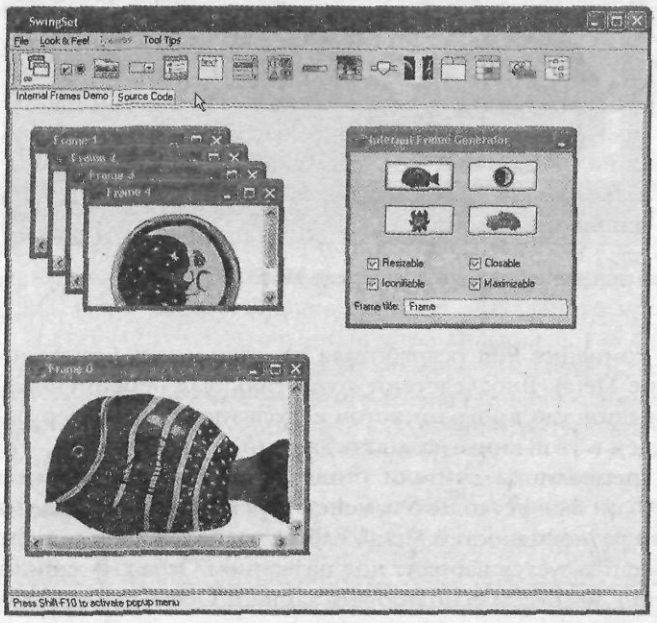


Рис. 7.1. Внешний вид окна программы в среде Windows

Объем данной книги не позволяет подробно описать, как расширять средства отображения пользовательских интерфейсов. Заметим лишь, что программисты сами могут изменить стиль интерфейса своих приложений и даже разработать новые инструменты для этого. Это длительный процесс, при котором нужно указать, как каждый компонент библиотеки Swing должен быть отображен на экране. Некоторые разработчики уже сделали это при переносе языка Java на такие нетрадиционные платформы, как портативные терминалы. Набор интересных реализаций средств отображения Swing можно найти по адресу `http://www.javootoo.com`.

В JDK 5.0 были реализованы новые средства отображения, которые получили название Synth и упрощают настройку компонентов для различных платформ. Работая с Synth, вы можете определить новый стиль, задавая лишь файлы изображений и XML-описания. Программировать при этом не нужно.

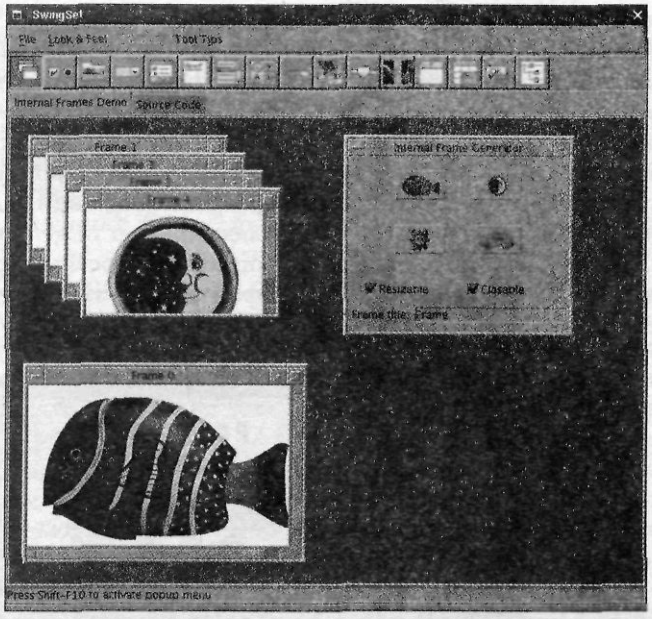


Рис. 7.2. Внешний вид окна программы в среде Motif

Более того, компания Sun разработала платформенно-независимый стиль, получивший название Metal. Впоследствии этот стиль был переименован в Java look and feel, однако большинство программистов используют старый термин. Мы последуем их примеру и будем в этой книге называть данный стиль Metal.

Некоторые специалисты считали стиль Metal слишком "тяжелым". Возможно, поэтому в JDK 5.0 он был несколько изменен (рис. 7.3). В настоящее время поддерживается несколько разновидностей Metal, слегка различающиеся цветами и шрифтами. По умолчанию используется вариант под названием Ocean. В данной книге, создавая компоненты Swing, мы будем использовать вариант Ocean стиля Metal.

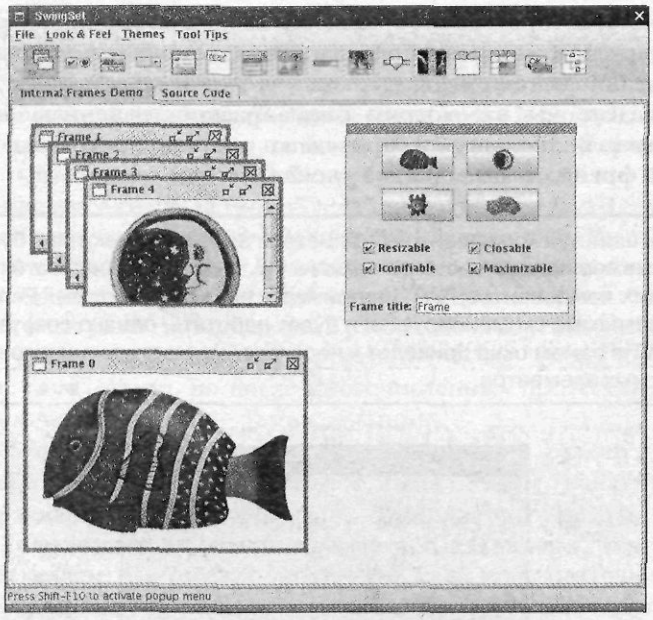


Рис. 7.3. Внешний вид программы в стиле Metal

В настоящее время большинство Java-приложений снабжаются интерфейсами, созданными на базе Swing. Исключением являются лишь программы, при разработке которых применялась интегрированная среда разработки Eclipse. В ней вместо Swing используется набор графических компонентов SWT, в котором, подобно AWT, компоненты отображаются платформено-ориентированными средствами. Информацию о SWT можно найти по адресу `http://www.eclipse.org/articles/`.

Если вы уже создавали графические пользовательские интерфейсы, используя для этого среды Visual Basic или C#, то знаете, насколько легко применять готовые компоненты, предоставляемые этими средами. Эти средства позволяют визуально программировать интерфейс и сами генерируют нужный для этого код. Хотя некоторые интегрированные среды для создания интерфейса на языке Java уже существуют, они еще не так развиты, как соответствующие средства визуального программирования в среде Windows. В любом случае, чтобы полностью понимать, как среда визуального программирования генерирует код для графического пользовательского интерфейса, и эффективно использовать нужные для этого средства, необходимо знать, как такой интерфейс создается вручную. Естественно, в большинстве случаев для этого нужно написать достаточно большой объем кода.

Создание фреймов

Окно верхнего уровня (т.е. окно, не содержащееся внутри другого окна) в языке Java называется *фреймом* (frame). В библиотеке AWT для такого окна предусмотрен класс `Frame`. В библиотеке `Swing` аналогом этого класса является `JFrame`. Класс `JFrame` расширяет класс `Frame` и представляет собой один из немногих компонентов

в библиотеке Swing, которые не отображаются на холсте. Кнопки, строка заголовков, пиктограммы и другие компоненты реализуются с помощью пользовательской оконной системы, а не библиотеки Swing.

В данном разделе мы рассмотрим самые распространенные способы работы с классом JFrame. В листинге 7.1 приведена простая программа, отображающая на экране пустой фрейм, показанный на рис. 7.4.

Имена большинства компонентов библиотеки Swing начинаются с буквы J. В качестве примеров можно привести классы JButton и JFrame. Они являются аналогами соответствующих компонентов AWT (например, Button и Frame). Если вы пропустите букву J, программа скомпилируется и будет работать, однако сочетание компонентов Swing и AWT в одном окне приведет к несогласованности внешнего вида и поведения интерфейсных элементов.

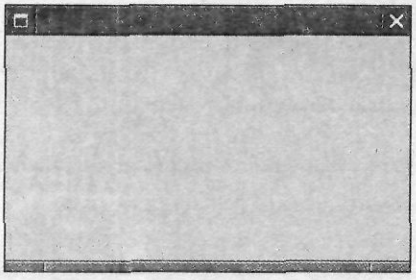


Рис. 7.4. Простейший фрейм

Листинг 7.1. Содержимое файла SimpleFrameTest.java

```
import javax.swing.*;

public class SimpleFrameTest
{
   public static void main(String[] args)
   {
      SimpleFrame frame = new SimpleFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

class SimpleFrame extends JFrame
{
   public SimpleFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}
```

Проанализируем эту программу построчно.

Классы библиотеки Swing находятся в пакете `javax.swing`. Имя `javax` означает, что данный пакет является расширением языка Java и не входит в число основных пакетов. Классы Swing относятся к расширению языка Java 1.1. Поскольку они не входят в иерархию основных классов языка, их можно загружать в браузеры, совместимые с версией Java 1.1. (Диспетчер защиты такого браузера не позволит загрузить пакет, имя которого начинается со слова "`java`".) Начиная с версии Java 2 Swing входит в иерархию основных пакетов. Классы Swing поддерживаются любой реализацией языка Java, совместимой с версией Java 2. Несмотря на это, имя `javax` в языке осталось, чтобы сохранить совместимость с версией Java 1.1. (На самом деле пакет Swing сначала назывался `com.sun.java.swing`, затем в ранних бета-версиях языка Java 2 это имя сократилось до `java.awt.swing`, потом средства Swing снова были возвращены в пакет `com.sun.java.swing`, но после многочисленных протестов программистов было выбрано окончательное имя – `javax.swing`.)

По умолчанию фрейм имеет совершенно бесполезный размер – 0×0 пикселей. В данном примере мы определяем подкласс `SimpleFrame`, конструктор которого устанавливает размер 300×200 пикселей. Выполнение метода `main()` класса `SimpleFrameTest` начинается с создания объекта `SimpleFrame`.

Далее мы определяем, что должно произойти, если пользователь закроет фрейм. В данном случае программа должна завершить свою работу. Для этого используется следующая строка кода:

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Если бы в программе использовалось несколько фреймов, завершать работу только потому, что пользователь закрыл один из них, было бы совершенно не обязательно. По умолчанию закрытый фрейм исчезает с экрана, а программа продолжает работу.

Создание фрейма не приводит к его автоматическому появлению на экране. В начале своего существования все фреймы невидимы. Это дает программисту возможность добавлять к фрейму компоненты еще до того, как он впервые появится на экране. Чтобы показать фрейм на экране, метод `main()` вызывает метод `setVisible()`.

На этом работа метода `main()` завершается. Обратите внимание на то, что это не приводит к прекращению работы программы. Завершается лишь ее основной поток. Поток пользовательского интерфейса, обеспечивающий работу программы, активизируется выводом фрейма на экран.

До появления JDK 5.0 для отображения фрейма на экране можно было использовать метод `show()`. Дело в том, что суперклассом `JFrame` является класс `Window`. У класса `Window` есть суперкласс `Component`, который также содержит метод `show()`. Начиная с JDK 1.2 метод `Component.show()` не рекомендован к использованию (deprecated), поэтому для того, чтобы сделать компонент видимым, мы рекомендуем вызывать метод `setVisible(true)`. Заметьте, что в JDK 1.4 и более ранних версиях ограничения на применение метода `Window.show()` не были установлены. Метод `show()` удобен тем, что он одновременно делает окно видимым и перемещает его на передний план. Увы, данное преимущество стало достоянием прошлого. В JDK 5.0 метод `show()` также переведен в разряд не рекомендованных к применению.

Окно выполняющейся программы показано на рис. 7.4. Как видите, такие элементы, как строка заголовка и пиктограммы для изменения размеров окна, отображаются

операционной системой, а не компонентами библиотеки Swing. При запуске этой программы в среде X Windows окно будет выглядеть совершенно по-другому. Все, что находится внутри фрейма, изображается с помощью библиотеки Swing. В данной программе фрейм просто заполняется фоном, цвет которого задается по умолчанию.

Работая с пакетом JDK 1.4, вы можете отключить все строки заголовка и другие элементы фреймов, вызвав метод `frame.setUndecorated(true)`.

Приведенная выше программа содержит два класса: один — для определения фрейма, а второй — для отображения фрейма на экране. Последний содержит метод `main()`. Такое решение вовсе не обязательно. Нередко можно увидеть метод `main()` в составе класса `Jframe`.

```
class SimpleFrame extends JFrame
{
   public static void main(String[] args)
   {
     SimpleFrame frame = new SimpleFrame();
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     frame.show();
   }

   public SimpleFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}
```

Если метод `main()` принадлежит классу фрейма, исходный код становится короче. Однако многие специалисты считают такой подход неприменимым. По их менению, предпочтительнее отделять класс, запускающий программу на выполнение, от класса, в котором определяется пользовательский интерфейс.

Позиционирование фреймов

В классе `JFrame` есть лишь несколько методов, позволяющих изменить внешний вид фрейма. Разумеется, благодаря наследованию можно использовать методы суперклассов класса `JFrame`, задающие размер и местоположение. К наиболее важным из них следует отнести следующие.

- Метод `dispose()`, закрывающий окно и освобождающий все системные ресурсы, использованные при его создании.
- Метод `setIconImage()`, получающий объект класса `Image` для применения в качестве пиктограммы при минимизации окна (в терминологии языка Java такое окно часто называют *свернутым* (iconized)).
- Метод `setTitle()`, позволяющий изменить текст в строке заголовка.
- Метод `setResizable()`, получающий в качестве параметра логическое значение и определяющий, имеет ли пользователь право изменять размеры фрейма.

Иерархия наследования для класса `JFrame` показана на рис. 7.5.

Как указано в документации на API, методы для изменения размеров и формы фреймов следует искать в классе `Component` (который является предком всех объектов графического пользовательского интерфейса) и в классе `Window` (суперклассе класса `Frame`). Например, метод `show()`, который используется для отображения фрейма на экране, находится в классе `Window`, а в классе `Component` есть метод `setLocation()`, позволяющий изменить местоположение компонента. В приведенном ниже методе левый верхний угол фрейма размещается в точке, находящейся на расстоянии `x` пикселей вправо и на расстоянии `y` пикселей вниз от точки (0,0) — левого верхнего угла экрана.

```
setLocation(x,y)
```

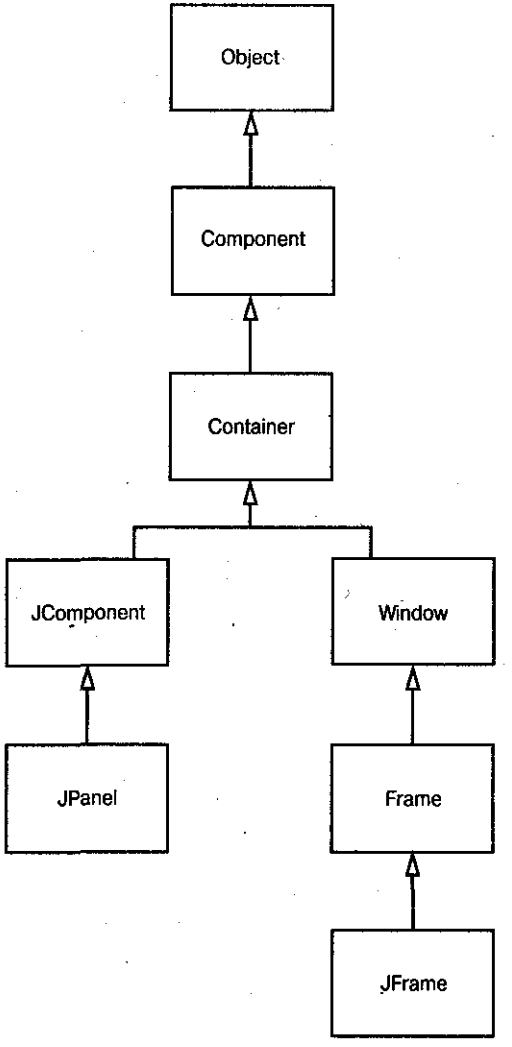


Рис. 7.5. Иерархия наследования для классов `JFrame` и `JPanel`

Аналогично, метод `setBounds()` класса `Component` позволяет одновременно изменить и размер, и местоположение компонента (в частности, объекта класса `JFrame`) с помощью следующего вызова:

```
setBounds(x, y, width, height)
```

Координаты фрейма, задаваемые методами `setLocation()` и `setBounds()`, вычисляются относительно экрана. Как будет показано в главе 9, координаты других компонентов внутри контейнера определяются относительно самого контейнера.

Запомните, если размер не задан явно, все фреймы по умолчанию имеют размер *0×0* пикселей. Чтобы не усложнять наш пример, мы задали размеры фреймов так, чтобы окно нормально отображалось на большинстве дисплеев. Однако в профессиональных приложениях сначала нужно проверить разрешающую способность экрана и написать код, изменяющий размер фрейма в соответствии с полученной величиной: окно, которое прекрасно смотрится на экране портативного компьютера, на экране дисплея с большой разрешающей способностью будет выглядеть, как почтовая марка. Как мы вскоре убедимся, получить размеры экрана в пикселях совсем несложно. Затем эту информацию можно использовать для вычисления оптимального размера окон в вашей программе.

В конце данного раздела описаны наиболее важные методы, позволяющие изменять внешний вид фреймов. Некоторые из них определены в классе `JFrame`. В процессе работы часто приходится выбирать наиболее подходящий метод для решения той или иной задачи. К сожалению, найти нужную информации в документации по JDK довольно сложно. Если речь идет о подклассах, в документацию помещается только информация о переопределяемых методах. Например, метод `toFront()` можно применять к объектам `JFrame`, но, поскольку он наследуется от класса `Window`, в документации по классу `JFrame` о нем ничего не сказано. Если вы считаете, что некий метод должен существовать, а в документации по классу, с которым вы работаете, его нет, попробуйте просмотреть описание методов *суперклассов* этого класса. В начале каждой страницы документации по API есть гипертекстовые ссылки на суперклассы. Кроме того, помимо описания новых и переопределенных методов приводится список всех наследуемых методов.

Для того чтобы продемонстрировать, что можно делать с окнами, закончим этот раздел простой программой, в которой устанавливаются размеры фреймов. При этом должны выполняться следующие условия.

- Фрейм должен занимать четверть экрана.
- Фрейм должен находиться в центре экрана.

Например, если экран имеет размер 800×600 пикселей, нужен фрейм размером 400×300 пикселей, верхний левый угол которого находится в точке (200, 150).

Для того чтобы определить размер экрана, надо выполнить следующие действия. Сначала следует вызвать статический метод `getDefaultToolkit()` класса `Toolkit`, который вернет объект `Toolkit`. (Класс `Toolkit` содержит много методов, предназначенных для взаимодействия с оконной системой конкретной платформы.) Затем надо вызвать метод `getScreenSize()`, который вернет размер экрана в виде объекта `Dimension`. Объект `Dimension` содержит ширину и высоту в общедоступных (!)

переменных width и height. Ниже представлен фрагмент кода, с помощью которого можно определить размер экрана.

```
Toolkit kit = Toolkit.getDefaultToolkit();
Dimension screenSize = kit.getScreenSize();
int screenWidth = screenSize.width;
int screenHeight = screenSize.height;
```

Мы также предоставим пиктограмму для фрейма. Поскольку процесс изображения рисунков на экране также зависит от операционной системы, для загрузки рисунка нам снова нужен объект Toolkit. Затем этот рисунок устанавливается в качестве пиктограммы.

```
Image img = kit.getImage("icon.gif");
setIcon(img);
```

В зависимости от операционной системы эта пиктограмма выводится по-разному. Например, в системе Windows пиктограмма отображается в левом верхнем углу окна, и ее можно увидеть в списке активных задач, если нажать клавиши <Alt+Tab>.

В листинге 7.2 приведен полный код программы. Запустив ее, обратите внимание на пиктограмму Core Java.

Обычно главный фрейм программы имеет максимальный размер. Для того чтобы максимально увеличить размер фрейма, работая с пакетом JDK 1.4, достаточно вызвать метод

```
frame.setExtendedState(Frame.MAXIMIZED_BOTH);
```

Если написанное вами приложение работает с несколькими дисплеями, для того чтобы определить размеры экранов, следует применять классы GraphicsEnvironment и GraphicsDevice. Начиная с JDK 1.4 можно также выполнять приложения в полноэкранном режиме.

Листинг 7.2. Содержимое файла CenteredFrameTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CenteredFrameTest
{
   public static void main(String[] args)
   {
      CenteredFrame frame = new CenteredFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

class CenteredFrame extends JFrame
{
   public CenteredFrame()
   {
```

```
        // Определение размеров экрана.

        Toolkit kit = Toolkit.getDefaultToolkit();
        Dimension screenSize = kit.getScreenSize();
        int screenHeight = screenSize.height;
        int screenWidth = screenSize.width;

        // Размещение фрейма по центру экрана.

        setSize(screenWidth / 2, screenHeight / 2);
        setLocation(screenWidth / 4, screenHeight / 4);

        // Установка пиктограммы и заголовка окна.

        Image img = kit.getImage("icon.gif");
        setIconImage(img);
        setTitle("CenteredFrame");
    }
}
```

API java.awt.Component 1.0

- `boolean isVisible()`

  Проверяет, является ли данный компонент видимым. Компоненты являются видимыми изначально, за исключением таких элементов верхнего уровня, как `JFrame`.

- `void setVisible(boolean b)`

  Отображает или скрывает компонент в зависимости от значения параметра `b` (`true` или `false`).

- `boolean isShowing()`

  Проверяет, можно ли отобразить компонент на экране. Для этого он должен быть видимым и находиться внутри контейнера, отображение которого также разрешено.

- `boolean isEnabled()`

  Проверяет, доступен ли компонент. Доступный компонент может получать информацию с клавиатуры. Компоненты являются доступными изначально.

- `void setEnabled(boolean b)`

  Разрешает или запрещает доступ к объекту.

- `Point getLocation()` **1.1**

  Восстанавливает местоположение левого верхнего угла данного компонента по отношению к левому верхнему углу содержащего его контейнера. (Объект `p` класса `Point` инкапсулирует координаты *x* и *y* в полях `p.x` и `p.y`.)

- `Point getLocationOnScreen()` **1.1**

  Возвращает местоположение левого верхнего угла компонента в экранной системе координат.

- `void setBounds(int x, int y, int width, int height)` **1.1**

  Перемещает компонент и изменяет его размеры. Расположение левого верхнего угла задают параметры `x` и `y`, а новый размер — параметры `width` и `height`.

- `void setLocation(int x, int y)` **1.1**
- `void setLocation(Point p)` **1.1**

  Перемещает компонент в новую точку. Если компонент не принадлежит верхнему уровню, координаты `x` и `y` (или `p.x` и `p.y`) отсчитываются относительно контейнера, в противном случае (например, для объектов класса `JFrame`) используется экранная система координат.

- `Dimension getSize(int width, int height)` **1.1**

  Возвращает текущий размер компонента.

- `get setSize(Dimension d)` **1.1**

  Задает новые размеры компонента.

java.awt.Window 1.0

- `void toFront()`

  Выводит окно на экран поверх других окон.

- `void toBack()`

  Помещает данное окно позади всех окон, выведенных на экран, и соответственно переупорядочивает остальные окна.

java.awt.Frame 1.0

- `void setResizable(boolean b)`

  Определяет, может ли пользователь изменять размеры фрейма.

- `void setTitle(String s)`

  Отображает в заголовке фрейма строку `s`.

- `void setIconImage(Image image)`

  Параметры `image` Изображение, которое вы хотите использовать в качестве пиктограммы для фрейма.

- `void setUndecorated(boolean b)` **1.4**

  Если параметр `b` имеет значение `true`, строка заголовка, кнопки изменения размеров и другие подобные элементы фрейма не отображаются.

- `boolean isUndecorated()` **1.4**

  Возвращает значение `true`, если строка заголовка, кнопки изменения размеров и другие подобные элементы фрейма не отображаются.

- `int getExtendedState()` **1.4**
- `void setExtendedState(int state)` **1.4**

  Определяет или устанавливает состояние окна. Переменная `state` может принимать следующие значения:

```
Frame.NORMAL
Frame.ICONIFIED
Frame.MAXIMIZED_HORIZ
Frame.MAXIMIZED_VERT
Frame.MAXIMIZED_BOTH
```

java.awt.Toolkit 1.0

- `static Toolkit getDefaultToolkit()`

  Возвращает объект класса `Toolkit`, предусмотренный по умолчанию.

- `Dimension getScreenSize()`

  Определяет размеры экрана.

- `Image getImage(string filename)`

  Загружает изображение из файла `filename`.

Отображение информации на панели

В этом разделе мы покажем, как выводить информацию во фрейме. Например, в главе 3 мы выводили строку на консоль в текстовом режиме. Теперь мы выведем сообщение во фрейме, как показано на рис. 7.6.

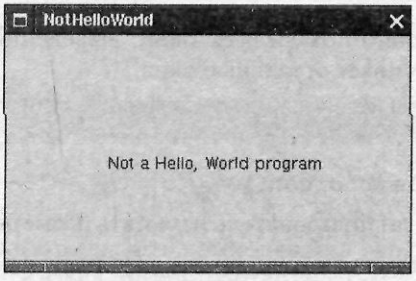


Рис. 7.6. Простая графическая программа

Строку сообщения можно вывести поверх фрейма, однако никто из профессиональных программистов так не делает. В языке Java фреймы предназначены именно для того, чтобы быть контейнерами для *компонентов* (например, меню или других элементов пользовательского интерфейса). Обычно рисунки выводятся в другом компоненте, называемом *панелью* (panel), которая добавляется к фрейму.

Оказывается, структура класса `JFrame` достаточно сложная. Его внутреннее устройство показано на рис. 7.7.

Как видно на рисунке, класс `JFrame` состоит из четырех слоев, каждый из которых представляет собой область (pane). Корневая область (root pane), область слоя (layered pane) и прозрачная область (glass pane) не представляют для нас интереса. Они нужны лишь для создания меню и области содержимого (content pane), выдержанных в определенном стиле. Наиболее интересной с точки зрения использования библиотеки Swing является область содержимого. При разработке фрейма компоненты добавляются в область содержимого с помощью следующего кода:

```
Container contentPane = frame.getContentPane();
Component c = . . .;
contentPane.add(c);
```

В JDK 1.4 и более старых версиях при попытке вызвать метод `add()` класса `Jframe` генерировалось *исключение с сообщением* `Do not use JFrame.add(). Use JFrame.getContentPane().add() instead.` В версии JDK 5.0 был реализован *метод* `JFrame.add()`, *который лишь переадресует вызов методу* `add()` *области* содержимого. Таким образом, в JDK 5.0 вы можете использовать следующий вызов.

```
frame.add().
```

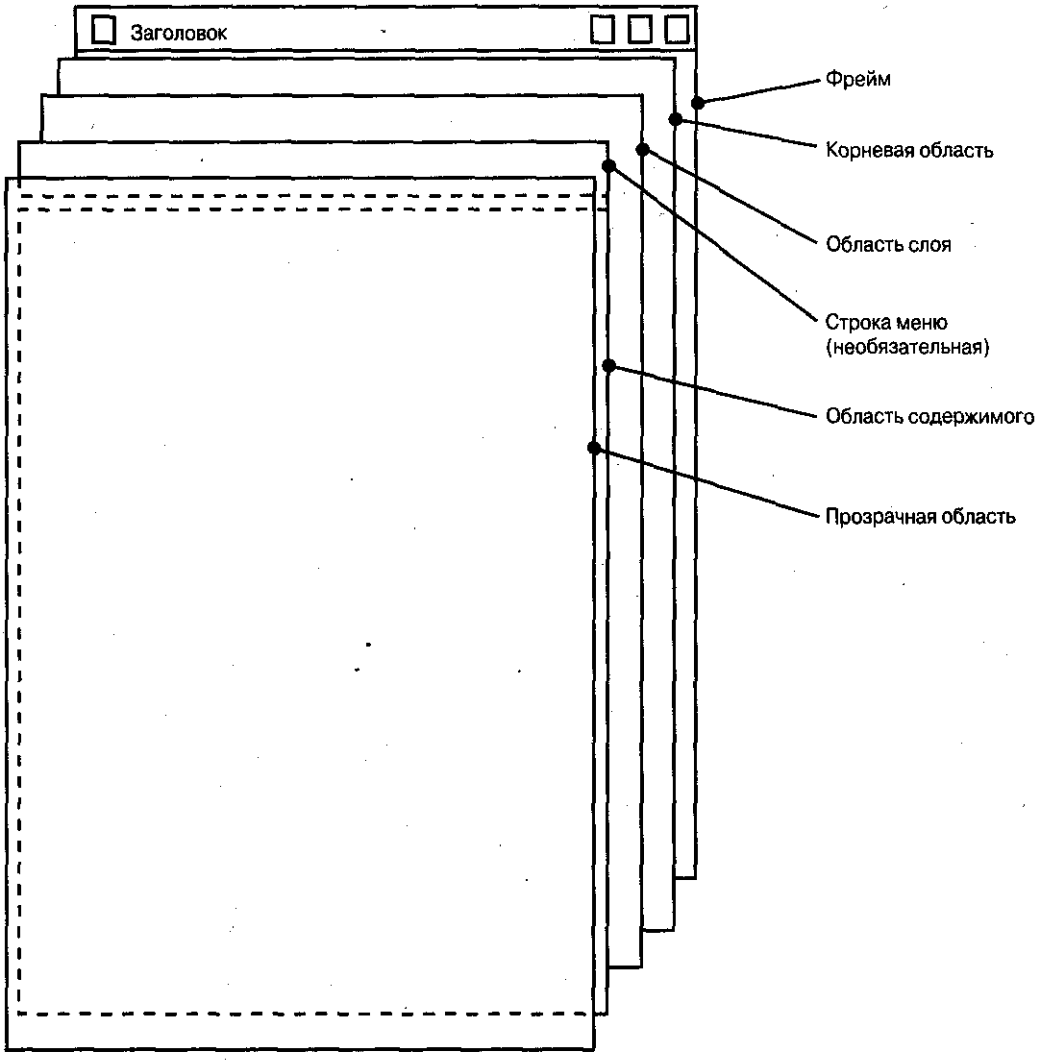


Рис. 7.7. Внутреннее устройство класса `JFrame`

В данном случае мы хотим добавить на панель содержимого одну *панель*, на которой мы отобразим наше сообщение. Панели реализуются классом `JPanel`. Они представляют собой элементы пользовательского интерфейса и обладают двумя полезными свойствами.

- На их поверхности можно рисовать.
- Они сами по себе являются контейнерами.

Таким образом, на панели можно размещать любые элементы пользовательского интерфейса (кнопки, линейные регуляторы и т.д.).

Чтобы модифицировать панель, создадим новый класс, используя механизм наследования, а затем переопределим или добавим методы, обеспечивающие дополнительные функциональные возможности.

В частности, для того, чтобы рисовать на панели, предварительно нужно выполнить следующие действия.

- Определить класс, расширяющий класс `JPanel`.
- Переопределить в этом классе метод `paintComponent()`.

На самом деле метод `paintComponent()` принадлежит классу `JComponent` — родительскому классу всех компонентов библиотеки Swing, не являющихся окнами. Он получает один параметр типа `Graphics`, который представляет собой набор установок для изображения рисунков и текста, например, в нем задается шрифт и цвет текста. Все рисование в языке Java осуществляется с помощью объектов класса `Graphics`. В этом классе предусмотрены методы для создания изображений и текста.

Параметр типа Graphics напоминает контекст устройства в среде Windows или графический контекст в среде X11.

Ниже демонстрируется процесс создания панели, на которой можно рисовать.

```
class MyPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      ... // Код, выполняющий рисование.
   }
}
```

Каждый раз, когда возникает необходимость перерисовать окно, неважно по какой причине, обработчик событий уведомляет об этом компонент. В результате выполняется метод `paintComponent()` всех компонентов.

Метод `paintComponent()` никогда не следует вызывать самостоятельно. Когда нужно перерисовать окно приложения, он вызывается автоматически и вмешиваться в этот процесс не следует.

В каких случаях необходима автоматическая перерисовка? Например, при увеличении или уменьшении размеров окна. Если пользователь открыл новое окно, которое перекрыло существующее, окно приложения оказывается поврежденным и также должно быть перерисовано. (Графическая система не сохраняет в памяти пиксели нижележащего окна.) Наконец, когда окно отображается на экране впервые, нужно выполнить код, указывающий, как и где должны отображаться его элементы.

Если вам нужно принудительно перерисовать экран, вызывайте вместо метода `paintComponent()` метод `repaint()`. Метод `repaint()` обратится к методу `paintComponent` каждого компонента и передаст ему сконфигурированный требуемым образом объект `Graphics`.

Как видно из приведенного выше фрагмента программы, метод `paintComponent()` имеет один параметр типа `Graphics`. Размеры объекта типа `Graphics` при отображении на экран задаются в пикселях. Координата (0,0) соответствует левому верхнему углу компонента, на поверхности которого выполняется рисование.

Отображение текста на экране считается особой разновидностью рисования. Класс `Graphics` содержит метод `drawString()`, который вызывается следующим образом:

```
g.drawString(text, x, y)
```

В нашем случае строка текста выводится на экран в исходном окне, занимающем приблизительно четверть экрана по ширине и половину — по высоте. Несмотря на то что нам пока неизвестно, как задается размер строки, установим ее начало в точке с координатами (75, 100). Это означает, что вывод первого символа строки начинается с точки, отстоящей на 75 пикселей от правого края и на 100 пикселей — от верхнего. (На самом деле 100 пикселей отсчитывается от верхнего края до базовой линии строки.) Итак, метод `paintComponent()` выглядит приблизительно следующим образом:

```
class NotHelloWorldPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
     ... // См. ниже.
     g.drawString("Not a Hello, World program",
                  Message_X, Message_Y);
   }

   public static final int MESSAGE_X = 75;
   public static final int MESSAGE_Y = 100;
}
```

Однако этот метод еще не закончен. Класс `NotHelloWorldPanel` расширяет класс `JPanel`, имеющий собственную информацию о том, каким образом следует изображать панель, т.е. какой цвет фона для этого нужно выбрать. Для того чтобы убедиться, что суперкласс действительно выполняет эту часть работы, перед началом рисования нужно вызвать метод `super.paintComponent()`.

```
class NotHelloWorldPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      ... // Код для рисования.
   }
}
```

Полностью код программы приведен в листинге 7.3.

Листинг 7.3. Содержимое файла NotHelloWorld.java

```
import javax.swing.*;
import java.awt.*;

public class NotHelloWorld
{
   public static void main(String[] args)
   {
      NotHelloWorldFrame frame = new NotHelloWorldFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм, содержащий панель с сообщением.
*/
class NotHelloWorldFrame extends JFrame
{
   public NotHelloWorldFrame()
   {
      setTitle("NotHelloWorld");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление панели к фрейму.

      NotHelloWorldPanel panel = new NotHelloWorldPanel();
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
   Панель, в составе которой отображается сообщение.
*/
class NotHelloWorldPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);

      g.drawString("Not a Hello, World program",
                   MESSAGE_X, MESSAGE_Y);
   }

   public static final int MESSAGE_X = 75;
   public static final int MESSAGE_Y = 100;
}
```

javax.swing.JFrame 1.2

- `Container getContentPane()`

  Возвращает область содержимого для объекта класса `JFrame`.

- `void add(Component c)`

  Добавляет указанный компонент в область содержимого фрейма. (В версиях JDK, предшествующих 5.0, попытка обратиться к этому методу приводила к генерации исключения.)

API `java.awt.Component 1.0`

- `void repaint()`

  Вызывает перерисовку компонента. Перерисовка выполняется сразу же после того, как возникнут условия, позволяющие сделать это.

- `public void repaint(int x, int y, int width, int height)`

  Вызывает перерисовку части компонента. Перерисовка выполняется сразу же после того, как возникнут условия, позволяющие сделать это.

API `javax.swing.JComponent 1.2`

- `void paintComponent(Graphics g)`

  В подклассах этот метод переопределяется. Содержимое метода определяет действия по выводу данных.

Двухмерные фигуры

Начиная с версии 1.0, в классе `Graphics` содержатся методы для рисования линий, прямоугольников, эллипсов и т.д. Однако эти операции предоставляют слишком ограниченный набор возможностей. Например, в них нельзя изменить толщину линии и повернуть фигуру на произвольный угол.

В версии 2.0 и последующих версиях JDK появилась библиотека *Java 2D*, реализующая более мощный набор графических операций. В этой главе мы рассмотрим лишь ее основные свойства – более подробная информация представлена во втором томе в главе, посвященной расширенным средствам AWT.

Для того чтобы нарисовать фигуру с помощью библиотеки Java 2D, нужно создать объект класса `Graphics2D`. Этот класс является подклассом класса `Graphics`. Если используется пакет JDK, содержащий библиотеку Java 2D, то такие методы, как `paintComponent()`, автоматически получают объекты класса `Graphics2d`. Нужно лишь применить приведение типов, как показано ниже.

```
public void paintComponent(Graphics g)
{
   Graphics g2 = (Graphics2D)g;
   ...
}
```

При создании геометрических фигур в библиотеке Java 2D применяется объектно-ориентированный подход. В частности, перечисленные ниже классы, предназначенные для рисования линий, прямоугольников и эллипсов, реализуют интерфейс `Shape`.

```
Line2D
Rectangle2D
Ellipse2D
```

Библиотека Java 2D поддерживает рисование и более сложных фигур, в частности, дуг, квадратичных и кубических кривых, а также произвольных линий. Более подробная информация об этом содержится в томе 2.

Для того чтобы нарисовать фигуру, сначала нужно создать объект некоторого класса, реализующего интерфейс `Shape`, а затем вызвать метод `draw()` класса `Graphics2D`.

```
Rectangle2D rect = ...;
g2.draw(rect);
```

До появления библиотеки Java 2D программисты использовали для рисования фигур методы класса `Graphics`, например метод `drawRectangle()`. На первый взгляд старый способ вызова методов выглядит немного проще. Однако, используя библиотеку Java 2D, программист имеет больше свободы действий — впоследствии он может уточнить свой рисунок с помощью многочисленных средств, предусмотренных в этой библиотеке.

Использование классов библиотеки Java 2D сопряжено с некоторыми трудностями. В отличие от методов рисования, предусмотренных в версии 1.0, в которых применялись целочисленные координаты, выраженные в пикселях, в библиотеке Java 2D используются координаты, представленные числами с плавающей точкой. Во многих случаях это очень удобно, поскольку часто программист формирует фигуру, координаты которой выражаются естественными единицами измерения (такими как дюймы или миллиметры), а уж затем переводит их в пиксели. В библиотеке Java 2D в ходе внутренних вычислений применяются числа с плавающей точкой с одинарной точностью. Этого вполне достаточно, ведь главной целью геометрических вычислений является вывод пикселей на экран или на принтер. Поскольку все округления в ходе вычислений не выходят за пределы одного пикселя, это совершенно не влияет на внешний вид фигуры. Более того, вычисления над числами с плавающей точкой с одинарной точностью на некоторых платформах выполняются быстрее, к тому же числа типа `float` занимают вдвое меньше памяти, чем числа типа `double`.

И все же манипулировать числами типа `float` иногда бывает неудобно, поскольку язык Java в вопросах приведения типов непреклонен и требует преобразовывать числа `double` в числа `float` явно. Рассмотрим, к примеру, следующее выражение:

```
float f = 1.2; // Ошибка.
```

Эта строка кода не будет скомпилирована, поскольку константа `1.2` имеет тип `double`, и компилятор зафиксирует потерю точности. Придется, формируя константу с плавающей точкой, указать суффикс `F`:

```
float f = 1.2F; // Правильно.
```

Теперь рассмотрим следующий пример:

```
rectangle2D r = ...
float f = r.getWidth(); // Ошибка.
```

Этот фрагмент не скомпилируется по тем же причинам. Метод `getWidth()` возвращает число, имеющее тип `double`. На этот раз нужно выполнить приведение типов:

```
float f = (float)r.getWidth(); // Правильно.
```

Поскольку указание суффиксов и приведение типов доставляют довольно много хлопот, разработчики библиотеки Java 2D решили предусмотреть *две версии* каждого класса для рисования фигур: в первой версии все координаты выражаются числами, имеющими тип `float` (для дисциплинированных программистов), а во второй — числами типа `double` (для ленивых). (В рамках этой книги мы относим себя ко второй *группе и везде используем тип* `double`.)

Разработчики библиотеки выбрали необычный и на первый взгляд запутанный способ создания пакетов, соответствующих этим двум версиям. Рассмотрим класс `Rectangle2D`. Это абстрактный класс, имеющий два конкретных подкласса, каждый из которых является внутренним и статическим.

```
Rectangle2D.Float
Rectangle2D.Double
```

На рис. 7.8 показана диаграмма наследования этих классов.

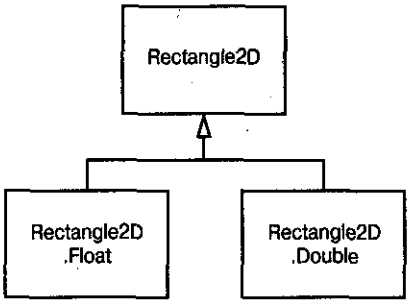


Рис. 7.8. Классы для рисования прямоугольника

Лучше всего совсем проигнорировать тот факт, что эти два конкретных класса являются внутренними и статическими, — это лишь уловка, чтобы избежать применения таких имен, как `FloatRectangle2D` и `DoubleRectangle2D`. (Статические внутренние классы детально рассмотрены в главе 6.)

При создании объекта класса `Rectangle2d.Float` используются координаты, имеющие тип `float`, а в объектах класса `Rectangle2d.Double` они выражаются числами типа `double`.

```
Rectangle2D.Float floatRect = new Rectangle2d.Float(10.0F,
   25.0F, 22.5F, 20.0F);
Rectangle2D.Double doubleRect = new Rectangle2d.Double(10.0,
   25.0, 22.5, 20.0);
```

Поскольку и класс `Rectangle2d.Float`, и класс `Rectangle2D.Double` расширяют один и тот же класс `Rectangle2D`, а методы в этих подклассах переопределяют методы суперкласса, нет нужды помнить точный тип координат, используемых при создании фигуры. Достаточно применять в качестве ссылок на прямоугольники переменные типа `Rectangle2D`.

```
Rectangle2D floatRect = new Rectangle2d.Float (10.0F,
   25.0F, 22.5F, 20.0F);
Rectangle2D doubleRect = new Rectangle2d.Double(10.0,
   25.0, 22.5, 20.0);
```

Таким образом, программисту приходится работать с внутренними классами лишь при создании объектов, описывающих фигуры.

Параметры конструктора задают координаты верхнего левого угла, ширину и высоту прямоугольника.

В действительности класс `Rectangle2D.Float` имеет один дополнительный метод, который не наследуется от класса `Rectangle2d`, — метод `setRect(float x, float y, float h, float w)`. Если вместо ссылок типа `Rectangle2D.Float` использовать ссылки типа `Rectangle2D`, этот метод станет недоступным. Однако это не большая потеря — класс `Rectangle2D` имеет свой метод `setRect()` с параметрами типа `double`.

И параметры, и значения, возвращаемые методами класса `Rectangle2D`, имеют тип `double`. Например, метод `getWidth()` возвращает значение типа `double`, даже если ширина хранится в переменной, имеющей тип `float`, в объекте класса `Rectangle2D.Float`.

Чтобы избежать значений типа `float`, используйте классы, в которых координаты выражаются числами типа `double`. Однако, если вам нужно создать тысячи объектов, следует подумать об экономии памяти. В этом вам помогут классы, в которых координаты задаются числами типа `float`.

Все, что было сказано нами о классах `Rectangle2D`, относится и к любым другим классам, предназначенным для рисования плоских фигур. Кроме того, существует еще и класс `Point2D` с подклассами `Point2D.Float` и `Point2D.Double`. Ниже приведен пример создания соответствующего объекта.

```
Point2D p = new Point2D.Double(10, 20);
```

Класс `point2D` очень полезен. Работа с объектами класса `Point2D` больше соответствует объектно-ориентированной парадигме, чем работа с отдельными координатами *x* и *y*. Многие конструкторы и методы получают на вход параметры, имеющие тип `Point2D`. Мы полагаем, что при малейшей возможности программист должен использовать именно эти объекты, так как они делают геометрические вычисления более понятными.

И класс `Rectangle2D`, и класс `Ellipse2D` являются производными классами одного и того же суперкласса `RectangularShape`. Всем известно, что эллипс не является прямоугольной фигурой, однако он ограничен прямоугольником (рис. 7.9).

В классе `RectangularShape` определено больше 20 методов, общих для всех фигур, среди которых находятся и такие полезные методы, как `getWidth()`, `getHeight()`, `getCenterX()` и `getCenterY()` (однако, к сожалению, во время создания этой книги в нем не было метода `getCenter()`, возвращающего центр фигуры в виде объекта класса `Point2D`).

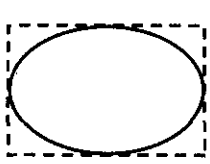


Рис. 7.9. Прямоугольник, ограничивающий эллипс

Отметим, что в новую библиотеку вошли и старые классы из библиотеки пакета JDK 1.0, заняв свое место в иерархии наследования. Так, например, классы `Rectangle` и `Point`, которые хранят координаты прямоугольника и его центра в виде целых чисел, расширяют классы `Rectangle2D` и `Point2D`.

На рис. 7.10 показаны отношения между классами, представляющими фигуры. Однако подклассы, в которых координаты выражаются с помощью чисел типа `float` и `double`, на рисунке не приведены. Классы, унаследованные от предыдущей версии, представлены закрашенными прямоугольниками.

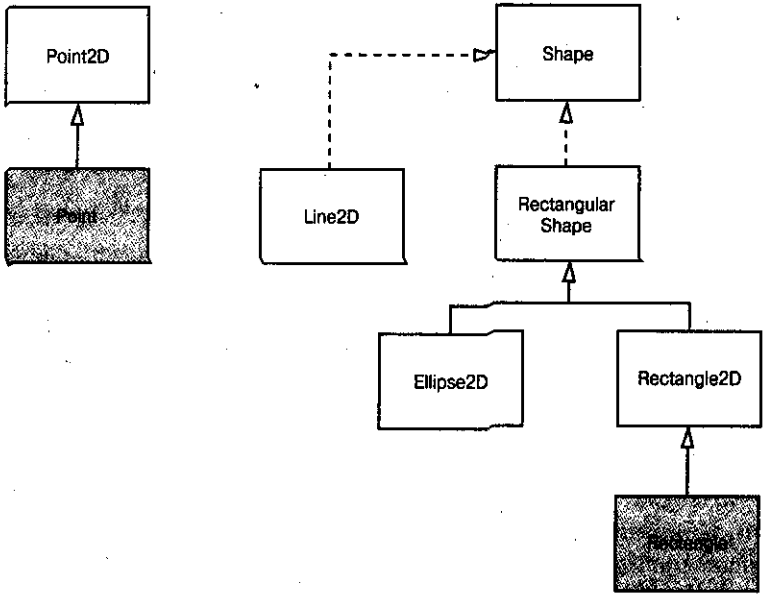


Рис. 7.10. Взаимосвязь между классами, представляющими геометрические фигуры

Объекты классов `Rectangle2D` и `Ellipse2D` создаются довольно легко. Достаточно указать следующие данные:

- координаты x и y левого верхнего угла;
- ширина и высота фигуры.

Для эллипсов эти параметры относятся к ограничивающим их прямоугольникам.

Например, приведенная ниже строка кода создает эллипс, ограниченный прямоугольником, левый верхний угол которого находится в точке (150, 200), а сам он имеет ширину и высоту, равные соответственно 100 и 50 пикселям.

```
Ellipse2D e = new Ellipse2D.Double(150, 200, 100, 50);
```

Иногда бывает нелегко определить, где должен находиться левый верхний угол фигуры. Довольно часто прямоугольник задается противоположными вершинами, но не обязательно они соответствуют левому верхнему и нижнему правому углу. В этом случае невозможно построить прямоугольник с помощью следующего выражения:

```
Rectangle2D rect = new Rectangle2D.Double(px, py,
   qx - px, qy - py); // Ошибка.
```

Если параметр p не соответствует координатам верхнего левого угла, одна или обе разности между координатами будут отрицательными и прямоугольник окажется пустым. В этом случае сначала нужно создать пустой прямоугольник, а затем использовать метод `setFrameFromDiagonal()`.

```
Rectangle2D rect = new Rectangle2D.Double();
rect.setFrameFromDiagonal(px, py, qx, qy);
```

Еще лучше, если противоположные вершины известны и представлены в виде объектов p и q класса `point2D`. В этом случае можно воспользоваться оператором

```
rect.setFrameFromDiagonal(p, q);
```

При создании эллипса обычно известны центр, ширина и высота ограничивающего прямоугольника, но не координаты его левого верхнего угла. Существует метод `setFrameFromCenter()`, использующий центр, но ему нужно задать одну из четырех угловых точек. Следовательно, эллипс лучше создать следующим образом:

```
Ellipse2D ellipse = new Ellipse2D.Double(centerX - width / 2,
   centerY - height / 2, width, height);
```

Для того чтобы нарисовать линию, нужно задать начальную и конечную точки, заданные либо как объекты класса `Point2D`, либо в виде пары чисел:

```
Line2D line = new Line2D.Double(start, end);
```

или

```
Line2D line = new Line2D.Double(startX, startY, endX, endY);
```

Программа, приведенная в листинге 7.4, рисует следующие фигуры.

- Прямоугольник.
- Эллипс, вписанный в прямоугольник.
- Диагональ прямоугольника.
- Окружность, центр которой совпадает с центром прямоугольника.

Результат работы программы показан на рис. 7.11.

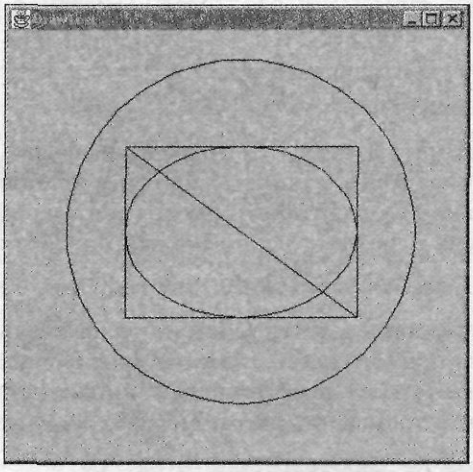


Рис. 7.11. Прямоугольники и эллипсы

Листинг 7.4. Содержимое файла `DrawTest.java`

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class DrawTest
{
   public static void main(String[] args)
   {
      DrawFrame frame = new DrawFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм, содержащий панель для рисования.
*/
class DrawFrame extends JFrame
{
   public DrawFrame()
   {
      setTitle("DrawTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление панели к фрейму.

      DrawPanel panel = new DrawPanel();
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 400;
}

/**
   Панель, на которой отображаются прямоугольники и эллипсы.
*/
class DrawPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;

      // Рисование прямоугольника.

      double leftX = 100;
      double topY = 100;
      double width = 200;
      double height = 150;

      Rectangle2D rect = new Rectangle2D.Double(leftX, topY,
         width, height);
      g2.draw(rect);

      // Рисование эллипса, вписанного в прямоугольник.
```

```
      Ellipse2D ellipse = new Ellipse2D.Double();
      ellipse.setFrame(rect);
      g2.draw(ellipse);

      // Рисование диагонали.

      g2.draw(new Line2D.Double(leftX, topY, leftX + width,
         topY + height));

      // Рисование окружности, центр которой совпадает
      // с центром прямоугольника.

      double centerX = rect.getCenterX();
      double centerY = rect.getCenterY();
      double radius = 150;

      Ellipse2D circle = new Ellipse2D.Double();
      circle.setFrameFromCenter(centerX, centerY,
         centerX + radius, centerY + radius);
      g2.draw(circle);
   }
}
```

java.awt.geom.RectangularShape 1.2

- `double getCenterX()`
- `double getCenterY()`
- `double getMinX()`
- `double getMinY()`
- `double getMaxX()`
- `double getMaxY()`

Возвращает координаты центра, наименьшую и наибольшую координату *x* и *y* описанного прямоугольника.

- `double getWidth()`
- `double getHeight()`

Возвращает ширину и высоту описанного прямоугольника.

- `double getX()`
- `double getY()`

Возвращает координаты *x* и *y* левого верхнего угла описанного прямоугольника.

java.awt.geom.Rectangle2D.Double 1.2

- `Rectangle2D.Double(double x, double y, double w, double h)`

Создает прямоугольник, заданный посредством координат верхнего левого угла, ширины и высоты.

java.awt.geom.Rectangle2D.Float 1.2

- `Rectangle2D.Float(float x, float y, float w, float h)`

Создает прямоугольник, заданный посредством координат верхнего левого угла, ширины и высоты.

java.awt.geom.Ellipse2D.Double 1.2

- `Ellipse2D.Double(double x, double y, double w, double h)`

  Создает эллипс, включающий прямоугольник которого имеет заданные координаты верхнего левого угла, ширину и высоту.

java.awt.geom.Point2D.Double 1.2

- `Point2D.Double(double x, double y)`

  Рисует точку с заданными координатами.

java.awt.geom.Line2D.Double 1.2

- `Line2D.Double(Point2D start, Point2D end)`
- `Line2D.Double(Point2D startX, Point2D startY, Point2D endX, Point2D startX)`

  Рисует линию с заданными координатами начала и конца.

Работа с цветом

Метод `setPaint()` класса `Graphics` позволяет выбрать цвет, который будет применяться при всех дальнейших операциях, связанных с рисованием в графическом контексте. Для того чтобы иметь возможность создавать многоцветные рисунки, следует выбрать некоторый цвет, нарисовать фигуру, затем выбрать новый цвет, нарисовать другую фигуру и т.д.

Цвет определяется с помощью класса `Color`. Класс `java.awt.Color` содержит константы, которые соответствуют 13 стандартным цветам, приведенным в табл. 7.1.

Ниже приведен пример выбора цвета.

```
g2.setPaint(Color.RED)
g2.drawString("Warning!", 100, 100);
```

Таблица 7.1. Стандартные цвета

BLACK	GREEN	RED
BLUE	LIGHT_GRAY	WHITE
CYAN	MAGENTA	YELLOW
DARK_GRAY	ORANGE	
GRAY	PINK	

До появления пакета JDK 1.4 имена констант, определяющих цвет, задавались строчными буквами, например `Color.red`. Это довольно странно, поскольку для констант принято формировать имена, состоящие из прописных букв. Начиная с версии JDK 1.4, стандартные названия цветов указываются как прописными, так и строчными буквами (для сохранения обратной совместимости).

Пользователь может указать свой произвольный цвет, создав объект класса Color из красной, зеленой и синей составляющих. В конструкторе класса Color компоненты цветов задаются целыми числами в диапазоне от 0 до 255.

```
Color(int redness, int greenness, int blueness)
```

Ниже приведен пример указания собственного цвета.

```
g2.setPaint(new Color(0, 128, 128); // Бледный сине-зеленый цвет.
g2.drawString("Welcome!", 75, 125);
```

Кроме сплошного цвета, можно выбирать более сложные конструкции с изменением оттенков. Подробно этот вопрос будет обсуждаться во втором томе в главе, посвященной расширенным средствам AWT. Если вместо Graphics2D используется объект класса Graphics, то для установки цвета нужно использовать метод setColor().

Для того чтобы установить цвет фона, следует применять метод setBackground() класса Component, являющегося предком класса JPanel.

```
MyPanel p = new MyPanel();
p.setBackground(Color.PINK);
```

Существует также метод setForeground(). Он задает цвет, который используется при рисовании компонента.

Методы brighter() и darker() класса Color создают (в полном соответствии со своими именами) более светлые или темные варианты текущего цвета. Используя метод brighter(), можно легко выделить заголовок или пункт меню. В действительности одного вызова метода brighter() для увеличения яркости недостаточно. Для того чтобы достичь заметного эффекта, желательно применить его трижды: c.brighter().brighter().brighter().

Для большинства известных цветов в классе SystemColor предусмотрены символьные имена. Константы в этом классе определяют цвета, которые используются для разнообразных элементов пользовательского интерфейса. Например, приведенный ниже вызов выбирает в качестве фона фрейма цвет, заданный про умолчанию для всех окон пользовательского интерфейса.

```
frame.setBackground(SyetemColor.window)
```

Фон заполняет окно при каждой перерисовке. Использовать цвета, заданные в классе SystemColor, особенно полезно, если вы хотите, чтобы пользовательский интерфейс был выдержан в той же цветовой гамме, что и рабочий стол. В табл. 7.2 приведены названия системных цветов и их описание.

java.awt.Color 1.0

- `Color(int r, int g, int b)`

  Создает объект класса Color.

Параметры	r	Интенсивность красного цвета (0–255)
	g	Интенсивность зеленого цвета (0–255)
	b	Интенсивность синего цвета (0–255)

Таблица 7.2. Системные цвета

desktop	Цвет фона для рабочего стола
activeCaption	Цвет фона для заголовков
activeCaptionText	Цвет текста для заголовков
activeCaptionBorder	Цвет рамки для неактивных заголовков
inactiveCaption	Цвет фона для неактивных заголовков
inactiveCaptionText	Цвет текста для неактивных заголовков
inactiveCaptionBorder	Цвет рамки для неактивных заголовков
window	Цвет фона для окна
windowBorder	Цвет рамки для окна
windowText	Цвет текста в окне
menu	Фон для меню
menuText	Цвет текста в меню
text	Цвет фона для текста
textText	Цвет текста
textInactiveText	Цвет текста для неактивных управляющих элементов
textHighlight	Цвет фона для выделенного текста
textHighlightText	Цвет выделенного текста
control	Цвет фона для управляющих элементов
controltext	Цвет текста для элементов управления
controlLtHighlight	Цвет для слабого выделения управляющих элементов
controlHighlight	Цвет для выделения управляющих элементов
controlShadow	Цвет тени для управляющих элементов
controlDkShadow	Темный цвет тени для управляющих элементов
scrollbar	Цвет фона для полосы прокрутки
info	Цвет фона для информационных сообщений
infoText	Цвет текста для информационных сообщений

java.awt.Graphics 1.0

- `void setColor(Color c)`

  Изменяет текущий цвет установки. При выполнении последующих графических операций будет использоваться новый цвет.

  Параметр `c` Новый цвет

java.awt.Graphics2D 1.0

- `void setPaints(Paint p)`

  Усанавливает атрибуты рисования для данного графического контекста. Класс `Color` реализует интерфейс `Paint`. Этот метод можно использовать для задания сплошного цвета при рисовании.

java.awt.Component 1.0

- `void setBackground(Color c)`

  Устанавливает цвет фона.

  Параметр `c` Новый цвет фона
- `void setForeground(Color c)`

  Устанавливает цвет переднего плана.

  Параметр `c` Новый цвет переднего плана

Заполнение фигур

Внутренность замкнутых фигур (например, прямоугольников или эллипсов) можно заполнять цветом. Чтобы сделать это, надо вместо метода `draw()` вызвать метод `fill()`.

```
Rectangle2D rect = ...;
g2.setPaint(Color.RED);
g2.fill(rect); // Заполняет объект rect красным цветом.
```

Программа, приведенная в листинге 7.5, заполняет прямоугольник красным цветом, а эллипс, вписанный в этот прямоугольник, — бледным сине-зеленым цветом (рис. 7.12).

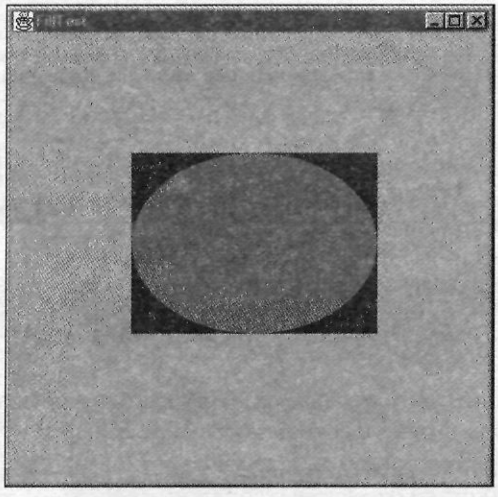


Рис. 7.12. Заполнение прямоугольника и эллипса

Листинг 7.5. Содержимое файла `FillTest.java`

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class FillTest
{
   public static void main(String[] args)
   {
      FillFrame frame = new FillFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм, содержащий панель для рисования.
*/
class FillFrame extends JFrame
{
   public FillFrame()
   {
      setTitle("FillTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление панели к фрейму.

      FillPanel panel = new FillPanel();
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 400;
}

/**
   Панель, на которой отображаются прямоугольник и эллипс.
*/
class FillPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;

      // Рисование прямоугольника.

      double leftX = 100;
      double topY = 100;
      double width = 200;
      double height = 150;

      Rectangle2D rect = new Rectangle2D.Double(leftX, topY,
         width, height);
      g2.setPaint(Color.RED);
```

```
      g2.fill(rect);

      // Рисование эллипса, вписанного в прямоугольник

      Ellipse2D ellipse = new Ellipse2D.Double();
      ellipse.setFrame(rect);
      // Бледный сине-зеленый цвет.
      g2.setPaint(new Color(0,  128, 128));
      g2.fill(ellipse);
   }
}
```

Использование шрифтов при выводе текста

Программа, приведенная в начале этой главы, выводила на экран строку текста, причем шрифт выбирался по умолчанию. Часто требуется, чтобы текст отображался разными шрифтами. Шрифты определяются *начертанием* символов. Имя начертания состоит из названия семейства шрифтов (например, Helvetica) и необязательного суффикса (например, Bold). Так, например, названия Helvetica и Helvetica Bold считаются частью одного и того же семейства Helvetica.

Для того чтобы выяснить, какие шрифты доступны на конкретном компьютере, вызовите метод `getAvailableFamilyNames()` класса `GraphicsEnvironment`. Этот метод возвращает массив строк, состоящих из названий всех доступных шрифтов. Чтобы создать экземпляр класса `GraphicsEnvironment`, описывающего графическую среду, используется статический метод `getLocalGraphicsEnvironment()`. Таким образом, приведенная ниже программа выводит названия всех шрифтов, доступных в вашей системе.

```
import java.awt.*

public class ListFonts
{
   public static void main(String[] args)
   {
      String[] fontNames = GraphicsEnvironment
         .getLocalGraphicsEnvironment()
         .getAvailableFontFamilyNames();
      for (int i = 0; i  fontNames.length; i++)
         System.out.println(fontNames[i]);
   }
}
```

В одной из систем список начинается такими строками:

```
Abadi MT Condenced light
Arial
Arial Black
Arial Narrow
Arioso
Baskerville
Binner Gothic
...
```

И так далее для следующих семидесяти шрифтов.

В документации на JDK утверждается, что такие суффиксы, как heavy, medium, oblique или gothic, определяют разновидности шрифтов в одном семействе. Как показывает наш опыт, это не так. Суффиксы Bold, Italic и Bold Italic распознаются как вариации шрифтов в одном и том же семействе, а остальные — нет.

К сожалению, не существует абсолютно надежного способа узнать, есть ли в компьютере шрифт, имеющий требуемый вид. Имена шрифтов представляют собой торговые марки, а сами шрифты могут быть защищены авторскими правами. Распространение шрифтов часто сопряжено с оплатой. Разумеется, аналогично дешевым подделкам дорогих духов, существуют имитации фирменных шрифтов. Например, имитация шрифта Helvetica распространяется с операционной системой Windows под именем Arial.

Для того чтобы поддерживать определенную систему, в AWT приняты пять *логических* названий шрифтов.

```
SansSerif
Serif
Monospaced
Dialog
DialogInput
```

Эти имена всегда соответствуют шрифтам, фактически существующим в данном клиентском компьютере. Например, в системе Windows шрифт SanSerif отображается в шрифт Arial.

Информация о соответствии шрифтов хранится в файле `fontconfig.properties`, который содержится в каталоге `jre/lib`. Информацию об этом файле вы можете найти по адресу `http://java.sun.com/j2se/5.0/docs/guide/intl/fontconfig.html`. В ранних версиях JDK для этой цели использовался файл `font.properties`.

Для того чтобы изобразить букву с помощью заданного шрифта, сначала необходимо создать объект класса `Font`, а затем указать название шрифта, его стиль и размер символов. Ниже показано, как можно создать объект класса `Font`.

```
Font helvb14 = new Font("Helvetica", Font.BOLD, 14);
```

Третий параметр задает размер шрифта. Для указания размеров используется единица измерений, которая называется пунктом (point). В одном дюйме содержится 72 пункта.

В конструкторе класса `Font` вместо фактического названия начертания можно использовать логическое имя шрифта. Затем нужно указать стиль, задав второй аргумент конструктора равным одному из следующих значений:

```
Font.PLAIN
Font.BOLD
Font.ITALIC
Font.BOLD + Font.ITALIC
```

В предыдущих версиях языка Java в качестве логических имен шрифтов использовались имена Helvetica, TimesRoman, Courier и ZapfDingbats. Для обратной совместимости эти имена все еще считаются логическими, несмотря на то, что имя Helvetica относится к реально существующему шрифту, а имена TimesRoman и ZapfDingbats и вовсе не являются названиями шрифтов — реальные шрифты называются "Times Roman" и "Zapf Dingbats" соответственно.

Начиная с версии JDK 1.3, можно использовать шрифты TrueType. Для этого нужно создать отдельный поток ввода для конкретного шрифта — обычно с файла на диске или с узла сети (потоки будут подробно рассматриваться в главе 12). Затем необходимо вызвать статический метод `FontCreateFont()`.

```
URL url = new URL("http://www.fonts.com/Wingbats.ttf");
InputStream in = url.openStream();
Font f = Font.createFont(Font.TRUETYPE_FONT, in);
```

Этот шрифт не является ни курсивным, ни полужирным, а его размер равен одному пункту. Чтобы получить шрифт заданного размера, используйте метод `deriveFont()`:

```
Font df = f.deriveFont(14.0F);
```

Существуют два переопределенных метода `deriveFont()`. Один из них (с параметром типа `float`) задает размер шрифта, другой (с параметром типа `int`) — стиль шрифта. Таким образом, вызов `f.deriveFont(14)` задает стиль, а не размер! (В результате будет установлен курсив, поскольку двоичное разложение числа 14 содержит единицу в разряде, соответствующем константе `ITALIC`.)

В языке Java шрифты состоят из букв, цифр, знаков пунктуации и некоторых других символов. Например, если вывести шрифтом Dialog символ `'\u2297'`, на экране появится знак ⊗. Доступными являются только символы, указанные в множестве Unicode.

Ниже приводится код, который выводит на экран строку "Hello, World" с помощью полужирного шрифта SanSerif размером 14 пунктов.

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
g2.setFont(sansbold14);
String message = "Hello, World!";
g2.drawString(message, 75, 100);
```

Теперь *выровняем строку по центру панели*. Для этого нужно знать ширину и высоту панели в пикселях. Процесс выравнивания зависит от трех факторов.

- Используемый шрифт (в данном случае — sans serif, полужирный, 14 пунктов).
- Символы строки (в данном случае — "Hello, World!").
- Устройство, на котором будет отображаться строка (в данном случае — экран).

Чтобы создать объект, представляющий характеристики экрана, нужно вызвать метод `getFontRenderContext()` класса `Graphics2D`. Он возвращает объект класса `FontRenderContext`. Этот объект нужно передать методу `getStringBounds()` класса `Font`.

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds(message, context);
```

Метод `getStringBounds()` возвращает прямоугольник, ограничивающий строку.

Чтобы выяснить, от чего зависят размеры этого прямоугольника, надо рассмотреть некоторые основные термины, применяемые при наборе текстов (рис. 7.13). *Базовая линия* (baseline) – это воображаемая линия, которая касается снизу таких символов, как, например, e. *Максимальная высота* (ascent) – максимальное расстояние от базовой линии до вершин *выступающих над базовой линией* символов (asceder), например, верхнего края буквы b или k. *Максимальная глубина посадки* символов (descent) – это расстояние от базовой линии до *нижнего края* (descender), например, нижнего края буквы p или g.

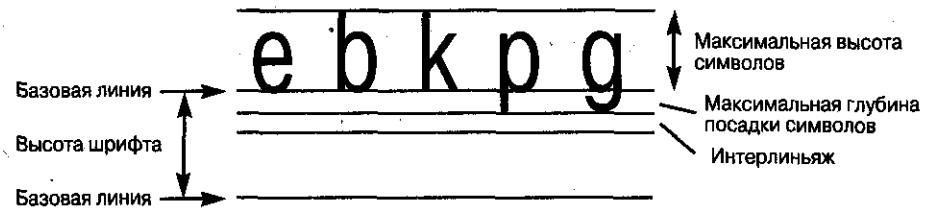


Рис. 7.13. Основные термины, применяемые при формировании строки текста

Интерлиньяж (leading) – это разность между максимальной глубиной посадки символов предыдущей строки и максимальной высотой символов следующей строки. *Высота* (height) *шрифта – это расстояние между соседними базовыми линиями*, т.е. максимальная высота и максимальная глубина посадки символов плюс интерлиньяж.

Ширина прямоугольника, возвращаемого методом `getStringBounds()`, задает горизонтальный размер строки. Высота прямоугольника равна сумме максимальной высоты и максимальной глубины посадки символов плюс интерлиньяж. Координата *y* отсчитывается от базовой линии. Для верхней части прямоугольника она является отрицательной величиной. Таким образом, ширину, высоту и максимальную высоту символов можно вычислить следующим образом:

```
double stringWidth = bounds.getWidth();
double stringHeight = bounds.getHeight();
double ascent = -bounds.getY();
```

Если нужно узнать, чему равен интерлиньяж или максимальная глубина посадки символов, следует вызвать метод `getLineMetrics()` класса `Font`. Этот метод возвращает объект класса `LineMetrics()`, в котором есть методы для определения указанных выше характеристик.

```
LineMetrics metrics = f.getLineMetrics(message, context);
float descent = metrics.getDescent();
float leading = metrics.getLeading();
```

В приведенном ниже фрагменте эта информация используется для выравнивания строки по центру содержащей ее панели.

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds(message, context);

// (x,y) — координаты верхнего левого угла строки текста
```

```
double x = (getWidth() - bounds.getWidth()) / 2;
double y = (getHeight() - bounds.getHeight()) / 2;

// Для определения базовой линии максимальная высота
// складывается с y
double ascent = -bounds.getY();
double baseY = y + ascent;
g2.drawString(message, (int) x, (int) baseY);
```

Для того чтобы понять, как происходит выравнивание текста по центру, отметим, что метод `getWidth()` возвращает ширину панели. Часть этой ширины занимает строка сообщения. Ее горизонтальный размер возвращает метод `bounds.getWidth()`. Оставшееся пространство нужно равномерно распределить по обеим сторонам строки. Следовательно, размер незаполненного пространства с каждой стороны должен равняться половине разности между шириной панели и длиной строки. Те же рассуждения справедливы и для выравнивания строки по высоте.

Итак, полученная программа рисует базовую линию и ограничивающий прямоугольник. На рис. 7.14 показано, как это выглядит на экране, а сама программа приведена в листинге 7.6.

Рис. 7.14. При выполнении программы отображаются ограничивающий прямоугольник и базовая линия строки

Листинг 7.6. Содержимое файла `FontTest.java`

```
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import javax.swing.*;

public class FontTest
{
   public static void main(String[] args)
   {
      FontFrame frame = new FontFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с панелью для вывода текста.
*/
```

```
class FontFrame extends JFrame
{
   public FontFrame()
   {
      setTitle("FontTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление панели к фрейму.

      FontPanel panel = new FontPanel();
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
   Панель, в центре которой выводится сообщение в рамке.
*/
class FontPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;

      String message = "Hello, World!";

      Font f = new Font("Serif", Font.BOLD, 36);
      g2.setFont(f);

      // Определение размеров сообщения.

      FontRenderContext context = g2.getFontRenderContext();
      Rectangle2D bounds = f.getStringBounds(message, context);

      // Определение координат левого верхнего угла
      // строки текста.

      double x = (getWidth() - bounds.getWidth()) / 2;
      double y = (getHeight() - bounds.getHeight()) / 2;

      // Для определения базовой линии используется
      // максимальная высота.

      double ascent = -bounds.getY();
      double baseY = y + ascent;

      // Вывод сообщения

      g2.drawString(message, (int) x, (int) baseY);

      g2.setPaint(Color.GRAY);

      // Вывод базовой линии.
```

```
      g2.draw(new Line2D.Double(x, baseY, x +
         bounds.getWidth(), baseY));

      // Вывод ограничивающего прямоугольника.

      Rectangle2D rect = new Rectangle2D.Double(x, y,
         bounds.getWidth(), bounds.getHeight());
      g2.draw(rect);
   }
}
```

java.awt.Font 1.0

- `Font(string name, int style, int size)`

  Создает новый объект класса `Font`.

  Параметры `name` Имя шрифта, представляющее собой либо имя начертания (например, Helvetica Bold), либо логическое название шрифта (например, Serif или SansSerif)

  `style` Стиль (`Font.PLAIN`, `Font.BOLD`, `Font.ITALIC` или `Font.BOLD+Font.ITALIC`)

  `size` Размер в пунктах (например, 12)

- `String getFontName()`

  Возвращает имя начертания (например, Helvetica Bold).

- `String getFamily()`

  Возвращает имя семейства шрифтов (например, Helvetica).

- `String getName()`

  Возвращает логическое имя (например, SansSerif), если оно было присвоено шрифту при его создании; в противном случае возвращает имя начертания.

- `Rectangle2D getStringBounds(String s, FontRenderContext)` **1.2**

  Возвращает прямоугольник, ограничивающий данную строку. Координата y прямоугольника отсчитывается от базовой линии. Координата y верхней части прямоугольника равна максимальной высоте со знаком "минус". Высота прямоугольника равна сумме максимальной высоты и максимальной глубины посадки. Ширина прямоугольника равна ширине строки.

- `LineMetrics getLineMetrics(String s, FontRenderContext context)` **1.2**

  Возвращает объект класса `LineMetrics`, позволяющий определить характеристики строки.

- `Font deriveFont(int style)` **1.2**
- `Font deriveFont(float size)` **1.2**
- `Font deriveFont(int style, float size)` **1.2**

  Возвращает новый объект класса `Font`, совпадающий с текущим, за исключением размера и стиля, заданных в качестве параметров.

java.awt.font.LineMetrics 1.2

- `float getAscent()`

  Определяет максимальную высоту — расстояние от базовой линии до верхней части прописных букв.

- `float getDescent()`

  Определяет максимальную глубину посадки — расстояние от базовой линии до нижней части букв.

- `float getLeading()`

  Определяет интерлиньяж — расстояние между нижней частью предыдущей строки и верхом следующей.

- `float getHeight()`

  Определяет общую высоту шрифта — расстояние между двумя базовыми линиями текста (максимальная высота и интерлиньяж плюс максимальная глубина посадки).

java.awt.Graphics 1.0

- `void setFont(Font font)`

  Устанавливает шрифт в графическом контексте. Этот шрифт будет использоваться в дальнейших операциях с графикой.

- `void drawString(String str, int x, int y)`

  Выводит на экран строку, шрифт и цвет которой определяются текущими установками графического контекста.

Параметры	`str`	Строка, подлежащая выводу
	`x`	Координата *x* начала строки
	`y`	Координата *y* начала строки

java.awt.Graphics2D 1.2

- `FontRenderContext getFontRenderContext()`

  Определяет контекст, который задает характеристики шрифта для выполнения операции с графикой.

- `void drawString(String str, float x, float y)`

  Выводит строку, шрифт и цвет которой определяются текущими установками.

Параметры	`str`	Строка, подлежащая выводу
	`x`	Координата *x* начала строки
	`y`	Координата *y* начала строки

Дополнительные операции с изображениями

Вы уже видели, как путем вывода линий и фигур создаются простые изображения. Сложные изображения, такие как фотографии, обычно создаются отдельно, например, в результате сканирования или применения графических редакторов. (Прочитав том 2, вы узнаете, что можно создавать изображения и помещать набор пикселей в массив. Эта процедура широко применяется при генерировании фрактальных изображений.)

Если изображения хранятся в файлах на диске компьютера или в глобальной сети, их можно прочитать, а затем отобразить на экране с помощью объектов класса `Graphics`. Пакет JDK 1.4 позволяет без труда организовать чтение изображений. Если рисунок хранится в локальном файле, достаточно использовать следующий фрагмент кода:

```
String filename = "...";
Image image = ImageIO.read(new File(filename));
```

При вызове метода можно также указать URL.

```
String filename = "...";
Image image = ImageIO.read(new URL(urlname));
```

Если рисунок не доступен, метод `read()` генерирует исключение `IOException`. Обработка исключений описана в главе 11. Пока будем считать, что наша простая программа перехватывает исключение и выводит на экран содержимое стека.

Теперь переменная `image` содержит ссылку на объект, инкапсулирующий графическое изображение. Этот объект можно отобразить на экране с помощью метода `drawImage()` класса `Graphics`.

```
public void paintComponent(Graphics g)
{
   ...
   g.drawImage(image, x, y, null);
}
```

Программа, приведенная в листинге 7.7, делает немного больше, многократно отображая указанное изображение. Результат работы программы показан на рис. 7.15. Вывод рисунка осуществляется с помощью метода `paintComponent()`. Сначала одна копия рисунка помещается в левый верхний угол окна, а затем вызывается метод `copyArea()`, который копирует его.

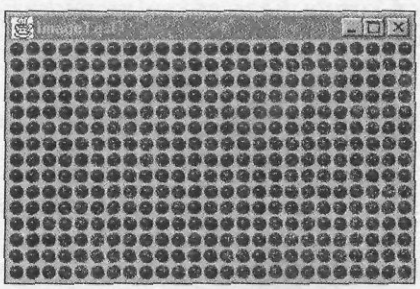


Рис. 7.15. Окно, содержащее копии одного и того же изображения

```
for (int i = 0; i * imageWidth , = getWidth(); i++)
   for (int j = 0; j * imageHeight <= getHeight(); j++)
     if ( i + j > 0)
        g.copyArea(0, 0, imageWidth, imageHeight,
           i * imageWidth, j * imageHeight);
```

Работая с JDK 1.3 или более ранними версиями, можно использовать класс `MediaTracker`. Объекты этого класса могут отслеживать процесс загрузки одного или нескольких изображений. (Имя "media" предполагает, что с помощью объектов этого класса можно также отслеживать загрузку аудио- и других медиа-файлов. Однако это — дело будущего, пока объекты класса `MediaTracker` могут работать только с изображениями.)

Передать изображение объекту класса `MediaTracker` можно с помощью следующих операторов:

```
MediaTracker tracker = new MediaTracker(component);
Image img = Toolkit.getDefaultToolkit().getImage(name);
int id = 1; // Идентификационный номер для отслеживания
            // процесса загрузки
tracker.addImage(img, id);
```

Одному объекту `MediaTracker` можно передавать сколько угодно изображений. Каждое изображение получает свой идентификационный номер, при этом допустима любая система нумерации. Для того чтобы приостановить выполнение программы до окончательного завершения загрузки изображения, используются следующие операторы:

```
try { tracker.waitForID(id); }
catch (InterruptedException e) {}
```

Если нужно загрузить несколько изображений, их все нужно передать объекту класса `MediaTracker`, а затем ожидать завершения загрузки. Сделать это можно с помощью приведенного ниже фрагмента кода.

```
try { tracker.waitForAll(); }
catch (InterruptedException e) {}
```

В листинге 7.7 приведен полный исходный текст программы для работы с изображениями. Этой программой мы завершим введение в программирование графики на языке Java. Более сложные приемы программирования будут изложены во втором томе.

Листинг 7.7. Содержимое файла `ImageTest.java`

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.imageio.*;
import javax.swing.*;

public class ImageTest
{
   public static void main(String[] args)
   {
      ImageFrame frame = new ImageFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

```
/**
   Фрейм с панелью, на которой выводится изображение.
*/
class ImageFrame extends JFrame
{
   public ImageFrame()
   {
      setTitle("ImageTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление панели к фрейму.

      ImagePanel panel = new ImagePanel();
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
   Панель, на которой многократно выводится изображение.
*/
class ImagePanel extends JPanel
{
   public ImagePanel()
   {
      // Получение изображения
      try
      {
         image = ImageIO.read(new File("blue-ball.gif"));
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      if (image == null) return;

      int imageWidth = image.getWidth(this);
      int imageHeight = image.getHeight(this);

      // Отображение рисунка в левом верхнем углу.

      g.drawImage(image, 0, 0, null);
      // Многократный вывод изображения на панели.

      for (int i = 0; i * imageWidth <= getWidth(); i++)
         for (int j = 0; j * imageHeight <= getHeight(); j++)
            if (i + j > 0)
               g.copyArea(0, 0, imageWidth, imageHeight,
```

```
                i * imageWidth, j * imageHeight);
   }

   private Image image;
}
```

API **javax.swing.ImageIO 1.4**

- `static BufferedImage read(File f)`
- `static BufferedImage read(URL u)`

Чтение изображения из указанного файла или копирование его с узла сети.

API **java.awt.Image 1.0**

- `Graphics getGraphics()`

Возвращает графический контекст для рисования в буфере изображений.

- `void flush()`

Освобождает ресурсы, занятые объектом изображения.

API **java.awt.Graphics 1.0**

- `boolean drawImage(Image img, int x, int y, ImageObserver observer)`

Выводит изображение. (Этот метод может вернуть управление до того, как изображение будет полностью выведено.)

Параметры	`img`	Изображение, предназначенное для вывода
	`x`	Координата *x* левого верхнего угла
	`y`	Координата *y* левого верхнего угла
	`observer`	Объект, обеспечивающий уведомление о ходе процесса (может быть нулевой ссылкой `null`)

- `boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)`

Выводит масштабированное изображение. Система масштабирует изображение для того, чтобы оно умещалось в области с заданной шириной и высотой. (Этот метод может вернуть управление до того, как изображение будет полностью выведено.)

Параметры	`img`	Изображение, предназначенное для вывода
	`x`	Координата *x* левого верхнего угла
	`y`	Координата *y* левого верхнего угла
	`width`	Требуемая ширина изображения
	`height`	Требуемая высота изображения
	`observer`	Объект, обеспечивающий уведомление о ходе процесса (может быть нулевой ссылкой `null`)

- `boolean copyArea(int x, int y, int width, int height, int dx, int dy)`

  Копирует область экрана.

Параметры		
	`x`	Координата *x* левого верхнего угла копируемой области
	`y`	Координата *y* левого верхнего угла копируемой области
	`width`	Ширина исходной области
	`height`	Высота исходной области
	`dx`	Расстояние по горизонтали от копируемой области до области, в которую выполняется копирование
	`dy`	Расстояние по вертикали от копируемой области до области, в которую выполняется копирование

- `void dispose()`

  Уничтожает графический контекст и освобождает ресурсы операционной системы. Все графические контексты, полученные при вызовах метода `Image.getGraphics()`, должны быть уничтожены, однако это не касается контекстов, обрабатываемых методом `paintComponent()`.

java.awt.Component 1.0

- `Image createImage(int width, int height)`

  Создает буфер изображения, используемый для двойной буферизации.

Параметры		
	`width`	Ширина изображения
	`height`	Высота изображения

java.awt.MediaTracker 1.0

- `MediaTracker(Component c)`

  Создает объект для контроля изображений, которые выводятся в составе указанного компонента

- `void addImage(Image image, int id)`

  Добавляет изображение в список контролируемых. После этого начинается процесс его загрузки.

Параметры		
	`image`	Контролируемое изображение
	`id`	Идентификатор, используемый для ссылки на изображение

- `void waitForID(int id)`

  Ожидает, пока изображение с указанным идентификационным номером будет загружено.

- `void waitForAll()`

  Ожидает, пока все контролируемые изображения будут загружены.

Глава 8

Обработка событий

В этой главе...

- Общие сведения об обработке событий
- Иерархия событий библиотеки AWT
- Семантические и низкоуровневые события в библиотеке AWT
- Типы низкоуровневых событий
- Действия
- Многоадресная передача событий
- Реализация источников событий

Обработка событий имеет огромное значение для создания программ с графическим пользовательским интерфейсом. Чтобы профессионально реализовать интерфейс программы, необходимо очень хорошо разбираться в этом механизме. В этой главе описана модель событий Java AWT. Вы узнаете, как перехватывать события мыши и клавиатуры, а также использовать простейшие элементы графического пользовательского интерфейса, например кнопки. В частности, здесь обсуждается обработка событий, генерируемых этими компонентами. В следующей главе будет показано, как объединить в одно целое компоненты библиотеки Swing, учитывая весь спектр событий, которые ими генерируются.

В Java 1.1 была предложена новая модель обработки событий, позволяющая создавать эффективно работающие пользовательские интерфейсы. Устаревшая модель события, использованная в Java 1.0, рассматриваться здесь не будет.

Общие сведения об обработке событий

Каждая операционная система, поддерживающая графический пользовательский интерфейс, непрерывно отслеживает такие события, как нажатие клавиш или щелчок мыши, а затем сообщает о них выполняемой программе. Программа решает, как реагировать на эти события (если это предусмотрено ее кодом). В языках, подобных языку Visual Basic, соответствие между событиями и кодом очевидно: среда программирования создает код, реагирующий на определенное событие, и помещает его в так называемую *процедуру обработки события* (event procedure). Например, после щелчка

на кнопке HelpButton будет вызвана процедура HelpButton_Click. Каждому компоненту пользовательского интерфейса в языке Visual Basic соответствует фиксированное множество событий, состав которого изменить невозможно.

Если для создания программы, реагирующей на события, используется язык, подобный языку C, программист должен создать код, непрерывно проверяющий очередь событий, которая передается ему операционной системой. (Обычно для этого код обработки событий помещается в гигантский цикл с большим количеством операторов switch.) Создавать такие программы чрезвычайно трудно. Однако подобный способ имеет преимущество, заключающееся в том, что множество событий ничем не ограничено, в отличие от языка Visual Basic, в котором очередь событий скрыта от программиста.

Подход, принятый в языке Java, представляет собой нечто среднее между подходами, применяемыми в языках Visual Basic и C. Он довольно сложен. При обработке событий, предусмотренных в библиотеке AWT, программист полностью контролирует передачу событий от *источников* (event sources) (например, кнопок или полос прокруток) к *обработчику* (event listener). *Любой* объект можно определить как обработчик некоего события — на практике все объекты так или иначе реагируют на события. Эта *модель делегирования событий* (event delegation model) позволяет достичь большей гибкости по сравнению с языком Visual Basic, в котором обработчики предопределены. Однако в процессе создания программы нужно написать намного больше кода, разобраться в котором намного сложнее (по крайней мере до тех пор, пока вы не привыкнете к этой технологии).

Источники событий содержат методы, позволяющие связывать их с обработчиками. Когда происходит соответствующее событие, источник извещает всех зарегистрированных обработчиков.

Поскольку язык Java является объектно-ориентированным, информация о событии инкапсулируется в *объекте* события (event object). В языке Java все события описываются подклассами java.util.EventObject. В качестве примеров можно привести ActionEvent и WindowEvent.

Различные источники могут порождать разные виды событий. Например, кнопка может создавать объекты класса ActionEvent, а окно — объекты класса WindowEvent.

Кратко механизм обработки событий AWT можно описать следующим образом.

- Обработчик представляет собой экземпляр класса, реализующего специальный интерфейс.
- Источник события — это объект, который может регистрировать обработчики и посылать им объекты событий.
- При наступлении события источник посылает объекты события всем зарегистрированным обработчикам.
- Обработчики используют информацию, инкапсулированную в объекте события, для того, чтобы решить, как реагировать на это событие.

Для регистрации обработчика применяется выражение в следующем формате:

объект_источника_события.add_*событие*_Listener(*обработчик_события*)

Рассмотрим следующий фрагмент программы:

```
ActionListener listener = ...;
JButton button = new JButton("OK");
button.addActionListener(listener);
```

Теперь обработчик оповещается о событии, связанном с кнопкой. Это может быть, например, простой щелчок.

Код, подобный приведенному выше, требует, чтобы класс обработчика реализовывал соответствующий интерфейс (в данном случае — `ActionListener`). Как и для остальных интерфейсов, реализация предполагает наличие методов, имеющих соответствующую сигнатуру. Для того чтобы реализовать интерфейс `ActionListener`, класс обработчика должен иметь метод с именем `actionPerformed()`, получающий в качестве параметра объект класса `ActionEvent`.

```
class MyListener implements ActionListener
{
   ...
   public void actionPerformed(ActionEvent event)
   {
      // Реакция на активизацию кнопки.
      ...
   }
}
```

Когда пользователь щелкает на кнопке, объект класса `JButton` создает объект класса `ActionEvent` и вызывает метод `listener.actionPerformed(event)`, передавая ему объект события. Обработчиками события (например, щелчка мышью на кнопке) можно назначать и несколько объектов одновременно. В этом случае после щелчка на кнопке объект класса `JButton` вызовет методы `actionPerformed()` всех зарегистрированных обработчиков.

На рис. 8.1 показано взаимодействие источника, обработчика и объекта события.

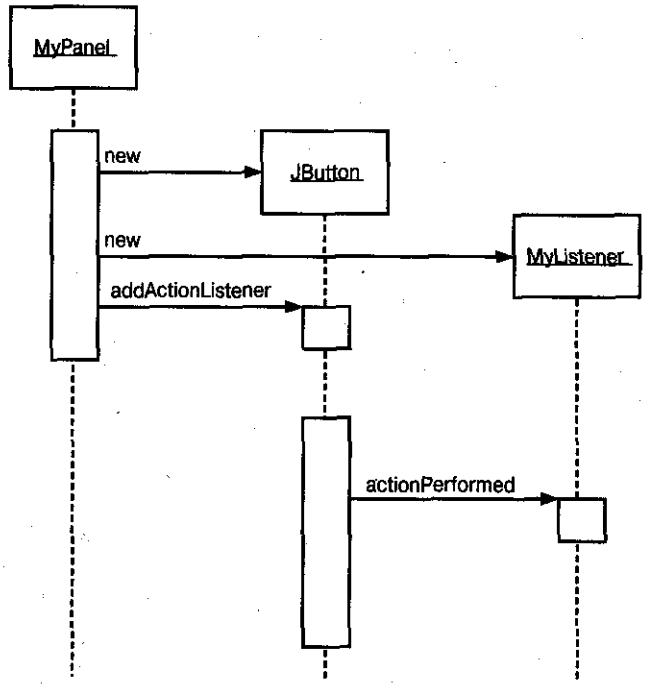


Рис. 8.1. Регистрация и обработка события

В этой главе особое внимание уделено обработке событий, связанных с *пользовательским интерфейсом*, таких как щелчок на кнопке или перемещение курсора мыши. Однако архитектура обработки событий не ограничивается поддержкой пользовательского интерфейса. Как вы увидите в следующей главе, оповещать обработчики могут и объекты, не имеющие никакого отношения к графическому интерфейсу. Обычно в таких случаях события генерируются при изменении свойств объекта.

Пример: обработка щелчка на кнопке

Для того чтобы лучше освоить модель делегирования событий, рассмотрим подробно, как обрабатывается щелчок на кнопке. Для этого нам понадобятся три кнопки, расположенные на панели, и три объекта-обработчика, обеспечивающие реакцию на события, связанные с кнопками.

В рамках этого сценария каждый раз, когда пользователь щелкает мышью на какой-нибудь кнопке, расположенной на панели, соответствующий обработчик получает объект класса `ActionEvent`. В нашей программе в ответ на щелчок будет изменяться цвет панели.

Прежде чем перейти к рассмотрению программы, реагирующей на щелчок кнопки, нужно объяснить, как создаются кнопки и как их размещают на панели. (Более подробно элементы графического пользовательского интерфейса рассматриваются в главе 9.)

Для того чтобы создать кнопку, в ее конструкторе нужно задать метки, пиктограмму или оба параметра одновременно. Рассмотрим две строки кода.

```
JButton yellowButton = new JButton("Yellow");
JButton blueButton = new JButton(new ImageIcon("blue-ball.gif"));
```

Кнопки помещаются на панель с помощью вызова метода с достаточно выразительным именем `add()`. Этот метод получает в качестве параметра указанный компонент и добавляет его в контейнер. Ниже приведен пример, демонстрирующий описанные действия.

```
class ButtonPanel extends JPanel
{
   public ButtonPanel()
   {
      JButton yellowButton = new JButton("Yellow");
      JButton blueButton = new JButton("Blue");
      JButton redButton = new JButton("Red");

      add(yellowButon);
      add(blueButton);
      add(redButton);
   }
}
```

Результат работы этого фрагмента показан на рис. 8.2.

Теперь, поместив кнопки на панель, надо добавить код, позволяющий отслеживать события, происходящие с этими кнопками. Для этого нужен класс, реализующий интерфейс `ActionListener`, в котором, как мы уже говорили, объявлен один метод — `actionPerformed()`. Сигнатура этого метода выглядит следующим образом:

```
public void actionListener(ActionEvent event)
```

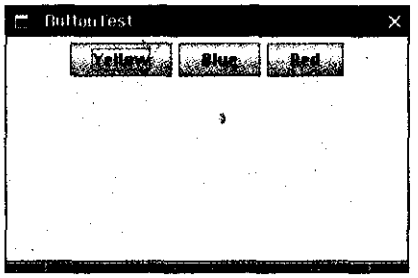


Рис. 8.2. Панель с кнопками

Интерфейс `ActionListener`, использованный в нашем примере, не ограничивается отслеживанием щелчков на кнопках. Он применяется во многих ситуациях.

- После двойного щелчка на пункте списка.
- После выбора пункта меню.
- После нажатия клавиши <Enter>, когда курсор находится в поле редактирования.
- По истечении заданного отрезка времени, отслеживаемого компонентом `Timer`.

Во всех случаях интерфейс `ActionListener` используется совершенно одинаково: метод `actionPerformed()` — единственный метод в интерфейсе `ActionListener` — получает в качестве параметра объект типа `ActionEvent`. Этот объект несет в себе информацию о событии.

Предположим, что мы хотим после щелчка на кнопке изменить цвет панели. Новый цвет нужно указать в классе обработчика.

```
class ColorAction implementsListener
{
   public ColorAction(Color c)
   {
      backgroundColor = c;
   }
   public void actionPerformed(ActionEvent event)
   {
      // Устанавливаем цвет панели.
      ...
   }

   private Color backgroundColor;
}
```

Затем мы создаем по одному объекту для каждого цвета и задаем их в качестве обработчиков событий, связанных с кнопкой.

```
ColorAction yellowAction = new ColorACtion(Color.YELLOW);
ColorAction blueAction = new ColorACtion(Color.BLUE);
ColorAction redAction = new ColorAction(Color.RED);

yellowButton.addActionListener(yellowAction);
blueButton.addActionListener(blueAction);
redButton.addActionListener(redAction);
```

Например, если пользователь щелкает на кнопке с надписью Yellow, вызывается метод actionPerformed() объекта yellowAction. Поле экземпляра backgroundColor хранит значение Color.YELLOW, поэтому выполняющийся метод может установить требуемый цвет панели.

Остался один небольшой вопрос. Объект ColorAction не имеет доступа к переменной panel. Эту проблему можно решить разными способами. Переменную panel можно указать в конструкторе класса. Но удобнее сделать класс ColorAction внутренним по отношению к классу ButtonPanel. В этом случае его методы получат доступ к внешним переменным автоматически. (Более подробно о внутренних классах см. в главе 6.)

Мы будем применять второй способ. Ниже показан пример включения класса ColorAction в класс ButtonPanel.

```
class ButtonPanel extends JPanel
{
   ...
   private class ColorAction implements ActionListener
   {
      ...
      public void actionPerformed(ActionEvent event)
      {
         setBackground(backgroundColor);
         // T.e. outer.setBackground(...).
      }
      private Color backgroundColor;
   }
}
```

Рассмотрим метод actionPerformed() более подробно. Класс ColorAction не имеет методов setBackground() и repaint(). Однако во внешнем классе ButtonPanel эти методы есть. Они вызываются в объекте ButtonPanel, создающем объекты внутреннего класса. (Напомним, что слово *outer* не является ключевым словом языка Java. Мы применяем его в качестве интуитивно понятного символа для обозначения ссылки на внешний класс в объекте внутреннего класса.)

Такая ситуация встречается довольно часто. Обработчики должны выполнять какие-то действия, влияющие на другие объекты. Часто бывает очень удобным заранее поместить класс обработчика в состав другого класса, который данный обработчик должен модифицировать.

Листинг 8.1 содержит полный код программы, в которой реализуется обработка событий. Как только пользователь щелкает на какой-нибудь кнопке, соответствующий обработчик изменяет цвет панели.

Листинг 8.1. Содержимое файла ButtonTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonTest
{
   public static void main(String[] args)
   {
      ButtonFrame frame = new ButtonFrame();
```

```
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм, содержащий панель с кнопками.
*/
class ButtonFrame extends JFrame
{
   public ButtonFrame()
   {
      setTitle("ButtonTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление панели к фрейму.

      ButtonPanel panel = new ButtonPanel();
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
   Панель с тремя кнопками.
*/
class ButtonPanel extends JPanel
{
   public ButtonPanel()
   {
      // Создание кнопок.

      JButton yellowButton = new JButton("Yellow");
      JButton blueButton = new JButton("Blue");
      JButton redButton = new JButton("Red");

      // Включение кнопок в состав панели.

      add(yellowButton);
      add(blueButton);
      add(redButton);

      // Создание обработчиков.

      ColorAction yellowAction = new ColorAction(Color.YELLOW);
      ColorAction blueAction = new ColorAction(Color.BLUE);
      ColorAction redAction = new ColorAction(Color.RED);

      // Связывание обработчиков с кнопками.

      yellowButton.addActionListener(yellowAction);
      blueButton.addActionListener(blueAction);
      redButton.addActionListener(redAction);
   }
```

```
   /**
      Обработчик, изменяющий цвет панели.
   */
   private class ColorAction implements ActionListener
   {
      public ColorAction(Color c)
      {
         backgroundColor = c;
      }

      public void actionPerformed(ActionEvent event)
      {
         setBackground(backgroundColor);
      }

      private Color backgroundColor;
   }
}
```

API javax.swing.JButton 1.2

- `JButton(String label)`

  Создает кнопку. Строка, передаваемая в качестве параметра, может представлять собой текст или, начиная с версии JDK 1.3, HTML-выражение, например `<HTML><B>OK</B></HTML>`

  Параметр `label` Текст на кнопке

- `JButton(Icon icon)`

  Создает кнопку.

  Параметр `icon` Пиктограмма на кнопке

- `JButton(String label, Icon icon)`

  Создает кнопку.

  Параметры `label` Текст на кнопке

  `icon` Пиктограмма на кнопке

API java.awt.Container 1.0

- `Component add(Component c)`

  Добавляет компонент c в контейнер.

API javax.swing.ImageIcon 1.2

- `ImageIcon(String filename)`

  Создает пиктограмму, изображение которой хранится в файле. Это изображение автоматически загружается в объект класса `MediaTracker` (см. главу 7).

Использование внутренних классов в качестве обработчиков событий

Отдельные программисты считают, будто внутренние классы, применяемые в качестве обработчиков, замедляют выполнение программы. Попробуем разобраться в этом. Для каждого компонента пользовательского интерфейса совсем не нужен новый класс. В нашем примере все три кнопки делят между собой один класс-обработчик. Разумеется, каждой из них соответствует отдельный объект этого класса. Однако эти объекты невелики. Такой объект содержит обозначение цвета и ссылку на панель.

Полагая, что время пользоваться внутренними классами уже наступило, мы настоятельно рекомендуем применять именно их в качестве обработчиков событий. В принципе для этого подойдут даже безымянные внутренние классы.

Приведем яркий пример, показывающий, как безымянные внутренние классы позволяют упростить исходный код программы. Вернемся к листингу 8.1. Обратите внимание на то, что обработка событий для всех кнопок совершенно одинакова и состоит из следующих операций.

1. Создание кнопки с соответствующей меткой.
2. Размещение кнопки на панели.
3. Создание обработчика, изменяющего цвет панели.
4. Связывание обработчика с объектом панели.

Чтобы упростить эту задачу, создадим вспомогательный метод.

```
void makeButton(String name, Color backgroundColor)
{
   JButton button = new JButton(name);
   add(button);
   ColorAction action = new ColorAction(backgroundColor);
   button.addActionListener(action);
}
```

Тогда конструктор класса `ButtonPanel` упрощается.

```
public ButtonPanel()
{
   makeButton("yellow", Color.YELLOW);
   makeButton("blue", Color.BLUE);
   makeButton("red", Color.RED);
}
```

Однако возможности упрощения программы этим не исчерпываются. Отметим, что класс `ColorAction` нужен лишь один раз: в методе `makeButton()`. Следовательно, его можно преобразовать в безымянный внутренний класс.

```
void makeButton(String name, final Color backgroundColor)
{
   JButton button = new JButton(name);
   add(button);
   button.addActionListener(new
      ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
```

```
            setBackgroiund(backgroundColor);
         }
      });
}
```

Код обработчика упростился. Метод `actionPerformed()` теперь просто ссылается на параметр `backgroundColor`. (Как и все локальные переменные, доступные во внутреннем классе, этот параметр должен быть терминальным.)

Явный конструктор теперь не нужен. Как мы уже видели в главе 6, механизм внутреннего класса автоматически генерирует конструктор, размещающий в памяти все локальные терминальные переменные, используемые в одном из методов этого класса.

На первый взгляд может показаться, что внутренние классы существенно усложняют код программы. Однако если вы научитесь выделять главное, абстрагируясь на время от остального кода, то легко освоите синтаксис внутренних классов. Рассмотрим пример.

```
button.addActionListener(new
      ActionListener()
      {
        public void actionPerformed(ActionEvent event)
        {
        setBackground(backgroundColor);
        }
      });
```

Поскольку обработчик события состоит всего лишь из нескольких операторов, мы полагаем, что в этом коде вполне можно разобраться, особенно, если программист не боится применять механизм внутренних классов.

В пакете JDK 1.4 реализован механизм, позволяющий создавать обработчики событий без программирования внутренних классов. Допустим, например, что приложение имеет кнопку Load, обработчик событий которой содержит всего один вызов метода:

```
frame.loadData();
```

Разумеется, можно было бы воспользоваться безымянным внутренним классом.

```
loadButton.addActionListener(new
          ActionListener()

      {
         public void actionPerformed(ActionEvent event)

         {

         frame.loadData();

         }

   };
```

Однако класс `EventHandler` может создать такой обработчик автоматически, вызвав метод `create()`.

```
EventHandler.create(ActionListener.class, frame, "loadData")
```

Разумеется, обработчик по-прежнему необходимо связать с источником событий.

```
loadButton.addActionListener((ActionListener)
   EventHandler.create(ActionListener.class, frame,   "loadData"));
```

Приведение типа необходимо, поскольку метод `create()` возвращает объект `Object`. Возможно, в последующих версиях JDK будут использованы универсальные типы, что упростит работу с данным методом.

Если обработчик событий вызывает метод с одним параметром, производным от обработчика событий, можно воспользоваться другой формой метода `create()`. Например, вызов

```
EventHandler.create(ActionListener.class, frame, "loadData",
      "source.text")
```

эквивалентен конструкции

```
new ActionListener()
{
   public void actionPerformed(ActionEvent event)
   {
      frame.loadData(
         ((JTextField)event.getSource()).getText());
   }
}
```

Обратите внимание на то, что обработчик событий превращает имена свойств `source` и `text` в вызовы методов `getSource()` и `getText()`, используя соглашения, принятые для компонентов JavaBeans. (Более полная информация о свойствах компонентов JavaBeans содержится во втором томе.)

Однако на практике эта ситуация встречается редко, и механизма поддержки параметров, не являющихся объектами классов, производных от классов событий, не существует.

Превращение компонентов в обработчики событий

Программист может совершенно свободно назначать обработчиком события, связанного с кнопкой, *любой* объект класса, реализующего интерфейс `ActionListener`. В нашем примере используются объекты нового класса, созданного специально для выполнения требуемых действий. Многие поступают иначе и добавляют метод `actionPerformed()` в существующий класс. Разумеется, этот класс также должен реализовывать интерфейс `ActionListener`. Например, можно легко превратить в обработчик класс `ButtonPanel`.

```
class ButtonPanel extends JPanel
   implements ActionListener
{
   ...
   public void actionPerformed(ActionEvents event)
   {
      // Устанавливает цвет фона.
```

```
        ...
    }
}
```

В этом случае панель сама себя назначает обработчиком событий для всех трех кнопок.

```
yellowButton.addActionListener(this);
blueButton.addActionListener(this);
redButton.addActionListener(this);
```

Отметим, что теперь три кнопки больше не имеют индивидуальных обработчиков. У них есть общий обработчик — панель, на которой они расположены. Следовательно, метод `actionPerformed()` должен как-то распознавать, на *какой* кнопке произведен щелчок.

Источник любого события распознается с помощью метода `getSource()` класса `EventObject` — суперкласса для всех событий. Источник события — это объект, генерирующий событие и уведомляющий об этом обработчик.

```
Object source = event.getSource();
```

После вызова данного метода `actionPerformed()` проверяет, какая из кнопок была источником события.

```
if (source == yellowButton) ...
else if (source == blueButton) ...
else if (source == redButton) ...
```

Разумеется, при этом нужно, чтобы все ссылки на кнопки хранились в полях содержащей их панели.

Некоторые программисты пользуются другим способом распознавания источника. В классе `ActionEvent` есть метод `getActionCommand()`, который возвращает *командную строку*, связанную с данным действием. Для кнопки он по умолчанию возвращает ее метку. Если вы предпочитаете такой способ, то метод `actionPerformed()` должен содержать следующий фрагмент кода (или подобный ему):

```
String command = event.getActionCommand();
if (command.equals("Yellow");) ...;
else if (command.equals("Blue")) ...;
else if (command.equals("Red")) ...;
```

Тем не менее полагаться на метки кнопок небезопасно. Легко сделать ошибку и назвать кнопку "Gray", а затем попытаться проверить источник события с помощью оператора

```
if (command.equals("Grey")) ...
```

Кроме того, при интернационализации вашей программы придется для каждого отдельного языка изменять название кнопки. Например, в немецком языке кнопки будут носить названия "Gelb", "Blau" и "Rot". В каждом из этих случаев программист будет вынужден изменять как метку кнопки, так и параметр метода `actionPerformed()`.

java.util.EventObject 1.1

- `Object getSource()`
  Возвращает ссылку на объект, являющийся источником события.

API `javax.swing.event.ChangeEvent 1.2`

- `String getActionCommand()`

  Возвращает командную строку, связанную с событием. Если событие произошло с кнопкой, командной строкой является метка строки (заметьте, что она может быть изменена с помощью метода `setActionCommand()`).

API `javax.swing.event.EventHandler 1.4`

- `static Object create(Class listenerInterface, Object target, String action)`
- `static Object create(Class listenerInterface, Object target, String action, String eventProperty)`
- `static Object create(Class listenerInterface, Object target, String action, String eventProperty, String listenerMethod)`

  Эти методы создают proxy-объект, реализующий заданный интерфейс. В результате либо указанный метод, либо все методы интерфейса выполняют заданное действие с целевым объектом.

  Параметр `action` может быть именем метода или свойством объекта. Если это свойство – выполняется метод `set...()`. Например, если параметр `action` имеет значение `text`, оно превращается в вызов метода `setText()`.

  Свойство события состоит из одного или нескольких имен свойств, разделенных точками. Первое свойство задает считывание параметра метода из обработчика, второе свойство – считывание результирующего объекта и т.д. Например, свойство `"source.text"` превращается в вызовы методов `getSource()` и `getText()`.

Пример: изменение стиля

По умолчанию программы, созданные с помощью пакета Swing, используют стиль Metal. Изменить стиль можно двумя способами. В подкаталоге JDK `jre/lib` можно предусмотреть файл `swing.properties` и задать в нем с помощью свойства `swing.defaultlaf` имя класса, определяющего нужный стиль, например:

```
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
```

Отметим, что стиль Metal описан в пакете `javax.swing`. Другие стили описаны в пакете `com.sun.java`. Они не являются обязательным атрибутом каждой реализации языка Java. В данный момент по соображениям, связанным с авторским правом, пакеты стилей для операционных систем Windows и Mac поставляются только с версиями JDK, *ориентированными на эти системы*.

Поскольку строки, начинающиеся символом #, в файлах, описывающих свойства стилей, игнорируются, в файле `swing.properties` можно предусмотреть несколько стилей, "закомментировав" ненужные.

```
#swing.defaulflaf=javax.swing.plaf.metal.MetalLookAndFeel
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
#swing.defaultlaf=com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

Чтобы изменить стиль интерфейса программы, ее нужно запустить заново. Программа прочитывает файл `swing.properties` лишь один раз, при запуске.

Для того чтобы изменять стиль динамически, вызовите метод `UIManager.setLookAndFeel()` и задайте ему имя требуемого стиля. Затем вызовите статический метод `SwingUtilities.updateComponentTreeUI()` и обновите с его помощью весь набор компонентов. Этому методу достаточно передать один компонент, остальные он найдет автоматически. Если нужный стиль не будет обнаружен или произойдет ошибка при его загрузке, метод `UIManager.setLookAndFeel` может сгенерировать различные исключения. Как обычно, вы можете не уделять внимания обработке исключений, поскольку эта тема будет полностью освещена в главе 11.

Рассмотрим пример, в котором задается стиль Motif графического интерфейса программы.

```
String plaf = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
try
{
   UIManager.setLookAndFeel(plaf);
   SwingUtilities.updateComponentTreeUI(panel);
}
catch(Exception e); { e.printStackTrace(); }
```

Для того чтобы пронумеровать инсталлированные реализации стилей, надо использовать следующее выражение:

```
UIManager.LookAndFeelInfo[] infos =
   UIManager.getInstalledLookAndFeels();
```

После этого вы можете с помощью приведенного ниже кода получить имя каждого стиля и реализующего его класса.

```
String name = infos[i].getName();
String className = infos[i].getClassName();
```

В листинге 8.2 приведена законченная программа, демонстрирующая переключение стилей (рис. 8.3). Эта программа очень похожа на ту, которая приведена в листинге 8.1. Следуя советам, изложенным выше, для того, чтобы задать действия с кнопками, т.е. переключение стилей, мы применили вспомогательный метод `makeButton()` и безымянный внутренний класс.

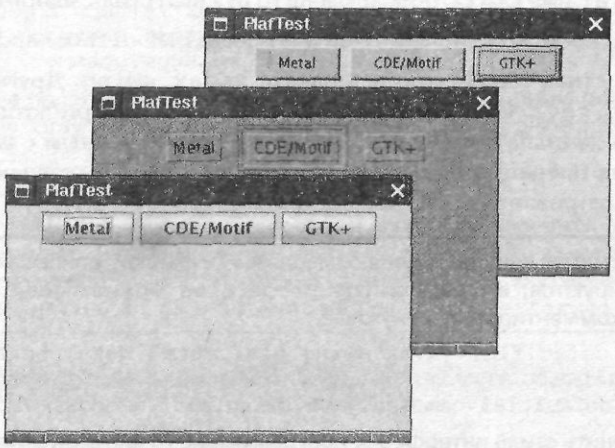


Рис. 8.3. Переключение стилей

У этой программы есть одно преимущество. Метод `actionPerformed()` во внутреннем классе-обработчике должен передать методу `updateComponentTreeUI()` ссылку `this` на объект внешнего класса `PlafPanel`. Напомним, что ссылка `this` на объект внешнего класса должна сопровождаться префиксом — именем этого класса.

```
SwingUtilities.updateComponentTreeUI(PlafPanel.this)
```

Листинг 8.2. Содержимое файла `PlafTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PlafTest
{
   public static void main(String[] args)
   {
      PlafFrame frame = new PlafFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с панелью, содержащей кнопки. С помощью
   кнопок изменяется стиль окна.
*/
class PlafFrame extends JFrame
{
   public PlafFrame()
   {
      setTitle("PlafTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление панели к фрейму.

      PlafPanel panel = new PlafPanel();
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
   Панель с кнопками, предназначенными для изменения стиля
*/
class PlafPanel extends JPanel
{
   public PlafPanel()
   {
      UIManager.LookAndFeelInfo[] infos =
         UIManager.getInstalledLookAndFeels();
      for (UIManager.LookAndFeelInfo info : infos)
         makeButton(info.getName(), info.getClassName());
   }
```

```
   /**
      Создает кнопки, изменяющие стиль отображения.
      @param name Имя кнопки
      @param plafName Имя класса, который описывает стиль
    */
   void makeButton(String name, final String plafName)
   {
      // Включение кнопки в состав панели.

      JButton button = new JButton(name);
      add(button);

      // Связывание обработчика событий с кнопкой.

      button.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               // Действие: переключение на другой стиль.
               try
               {
                  UIManager.setLookAndFeel(plafName);
                  SwingUtilities.updateComponentTreeUI
                     (PlafPanel.this);
               }
               catch(Exception e) { e.printStackTrace(); }
            }
         });
   }
}
```

API **javax.swing.UIManager 1.2**

- `static UIManager.LookAndFeelInfo[] getInstalledLookAndFeels()`

  Получение массива объектов, описывающих реализации инсталлированных стилей.

- `static setLookAndFeel(String className)`

  Установка текущего стиля.

  Параметр `className` Имя, класса, реализующего стиль

API **javax.swing.UIManager.LookAndFeelInfo 1.2**

- `String getName()`

  Возвращает имя стиля.

- `String getClassName()`

  Возвращает имя класса, реализующего стиль.

Пример: обработка событий, связанных с окнами

Не все события обрабатываются так же просто, как события, связанные с кнопками. Рассмотрим более сложный случай, о котором мы уже упоминали в главе 7. До появления `EXIT_ON_CLOSE` в пакете JDK 1.3 программисты должны были вручную реализовывать выход из программы при закрытии главного фрейма. Если программа представляет собой профессиональное приложение, она обязательно должна завершаться корректном, и пользователь должен иметь гарантию, что он не потеряет результаты выполненной работы. Например, если пользователь закрыл фрейм, можно вывести на экран диалоговое окно и предупредить его о необходимости сохранить результаты работы и только после его согласия завершить выполнение приложения.

Если пользователь пытается закрыть фрейм, объект `JFrame` становится источником события `WindowEvent`. Для того чтобы обработать это событие, нужен соответствующий объект, который следует зарегистрировать известным вам образом.

```
WindowListener listener = ...;
frame.addWindowListener(listener);
```

В данном случае обработчик должен быть экземпляром класса, реализующего интерфейс `WindowListener`. В интерфейсе `WindowListener` есть семь методов. Фрейм вызывает их в ответ на семь разных событий, которые могут произойти с окном. Их имена отражают их назначение. Исключением, может быть, является слово iconified, что в системе Windows означает "свернутое" окно. Вот как выглядит полный интерфейс `WindowsListener`.

```
public interface WindowListener
{
   void windowOpened(WindowEvent e);
   void windowClosing(WindowEvent e);
   void windowClosed(WindowEvent e);
   void windowIconified(WindowEvent e);
   void windowDeiconified(WindowEvent e);
   void windowActivated(WindowEvent e);
   void windowDeactivated(WindowEvent e);
}
```

Если нужно проверить, было ли окно увеличено до максимальных размеров, следует использовать обработчик типа `WindowStateListener`. Подробную информацию о нем можно найти в документации на API.

Как обычно, любой класс, реализующий какой-либо интерфейс, должен реализовывать *все его методы*. *В данном случае это означает создание тела семи методов.* Напомним, что нас интересует только один из них — метод `windowClosing()`.

Разумеется, можно определить класс, реализующий интерфейс `WindowListener`, добавить в метод `windowClosing()` вызов `System.exit(0)` и написать пустые функции для остальных шести методов.

```
class Terminator implements WindowListener
{
   public void windowClosing(WindowEvent e)
   {
      System.exit(0);
   }
```

```
   public void windowOpened(WindowEvent e) {}
   public void windowClosing(WindowEvent e) {}
   public void windowClosed(WindowEvent e) {}
   public void windowIconified(WindowEvent e) {}
   public void windowDeiconified(WindowEvent e) {}
   public void windowActivated(WindowEvent e) {}
   public void windowDeactivated(WindowEvent e) {}
}
```

Классы-адаптеры

Создавать код для шести методов, которые ничего не делают, — неблагодарное занятие. Чтобы упростить нашу задачу, для каждого из интерфейсов обработчиков, в котором содержится несколько методов, создается *класс-адаптер*, реализующий все эти методы, причем тело их пусто. Например, класс `WindowAdapter` содержит семь методов, не выполняющих никаких операций. Следовательно, класс-адаптер автоматически удовлетворит требованиям, предъявляемым к реализации соответствующего интерфейса. Программист может расширить класс-адаптер и уточнить нужные реакции на некоторые и даже на все типы событий, происходящих в интерфейсе. (Заметьте, что интерфейс `ActionListener` содержит только один метод, поэтому для него класс-адаптер не нужен.)

Рассмотрим применение адаптера для окна. Расширим класс `WindowAdapter`, унаследовав шесть пустых методов и заменив метод `windowClosing()`.

```
class Terminator extends WindowAdapter
{
   public void windowClosing(WindowEvent e)
   {
     System.exit(0);
   }
}
```

Теперь объект класса `Terminator` можно зарегистрировать как обработчик событий.

```
WindowListener listener = new Terminator();
frame.addWindowListener(listener);
```

С этого момента, как только фрейм сгенерирует событие, он передаст его обработчик, вызвав один из его семи методов (рис. 8.4). Шесть из этих методов ничего не делают, а метод `windowClosing()` выполняет вызов `System.exit(0)`, завершая работу приложения.

Если, расширяя класс адаптера, вы неправильно укажете имя метода, компилятор не сообщит об ошибке. Например, если в подклассе `WindowAdapter` вы создадите метод `windowIsClosing()`, то получите новый класс, содержащий восемь методов, причем метод `windowClosing()` не будет выполнять никаких действий.

Создание класса обработчика, расширяющего класс `WindowAdapter`, — несомненный шаг вперед, но можно достичь еще большего. Объекту обработчика не обязательно давать имя, достаточно написать следующее выражение:

```
frame.addWindowListener(new Terminator());
```

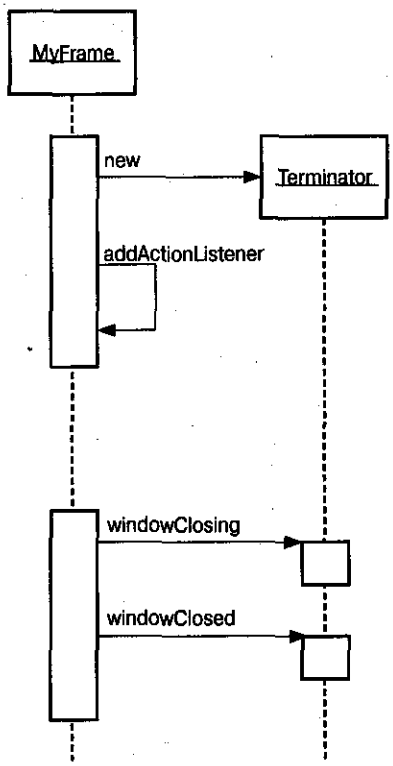


Рис. 8.4. Обработка событий, связанных с окном

Но и это еще не все! Можно использовать в качестве обработчика безымянный внутренний класс фрейма.

```
frame.addWindowListener(new
   WindowAdapter()
   {
      public void windowClosing(WindowEvent e)
      {
         System.exit(0);
      }
   } );
```

В этом фрагменте кода выполняются следующие действия.

- Определяется безымянный класс, расширяющий класс `WindowAdapter`.
- В этом безымянном классе переопределяется метод `windowClosing()`. Как и прежде, данный метод отвечает за выход из программы.
- Новый класс наследует от класса `WindowAdapter` шесть методов, не выполняющих никаких действий.
- Создается объект класса. Этот объект также не имеет имени.
- Безымянный объект передается методу `addWindowListener()`.

API java.awt.event.WindowListener 1.1

- `void windowOpened(WindowEvent e)`

  Этот метод вызывается после открытия окна.

- `void windowClosing(WindowEvent e)`

  Этот метод вызывается, когда пользователь отдает оконному диспетчеру команду закрыть окно. Обратите внимание на то, что окно закрывается, только если вызван метод `hide()` или `dispose()`.

- `void windowClosed(WindowEvent e)`

  Этот метод вызывается после закрытия окна.

- `void windowIconified(WindowEvent e)`

  Этот метод вызывается после свертывания окна.

- `void windowDeiconified(WindowEvent e)`

  Этот метод вызывается после развертывания окна.

- `void windowActivated(WindowEvent e)`

  Этот метод вызывается после активизации окна. Активным может быть только фрейм или диалоговое окно. Обычно оконный диспетчер специально отмечает активное окно, например подсвечивает его заголовок.

- `void windowDeactivated(WindowEvent e)`

  Этот метод вызывается после того, как окно стало неактивным.

API java.awt.event.WindowStateListener 1.4

- `void windowStateChanged(WindowEvent event)` **1.4**

  Этот метод вызывается после того, как окно было увеличено до максимальных размеров, свернуто или если были восстановлены его нормальные размеры.

API java.awt.event.WindowEvent 1.1

- `int getNewState()` **1.4**
- `int getOldState()` **1.4**

  Эти методы возвращают новое или старое состояние окна при изменении его состояния. Возвращаемое целое число может принимать одно из следующих значений:

```
Frame.NORMAL
Frame.ICONIFIED
Frame.MAXIMIZED_HORIZ
Frame. MAXIMIZED_VERT
Frame.MAXIMIZED_BOTH
```

Иерархия событий библиотеки AWT

После обсуждения механизма обработки событий перейдем к более общим вопросам. Как уже указывалось ранее, в языке Java при обработке событий используется объектный подход. Все события являются потомками класса `EventObject` из пакета `java.util`. (Общий суперкласс не назвали именем `Event`, так как это имя носил класс событий в старой модели. Хотя старая модель в настоящее время не рекомендована к использованию, ее классы все еще являются частью библиотеки Java.)

Класс `EventObject` имеет подкласс `AWTEvent`, являющийся родительским по отношению ко всем классам событий из пакета AWT. На рис. 8.5 показана диаграмма наследования событий из пакета AWT.

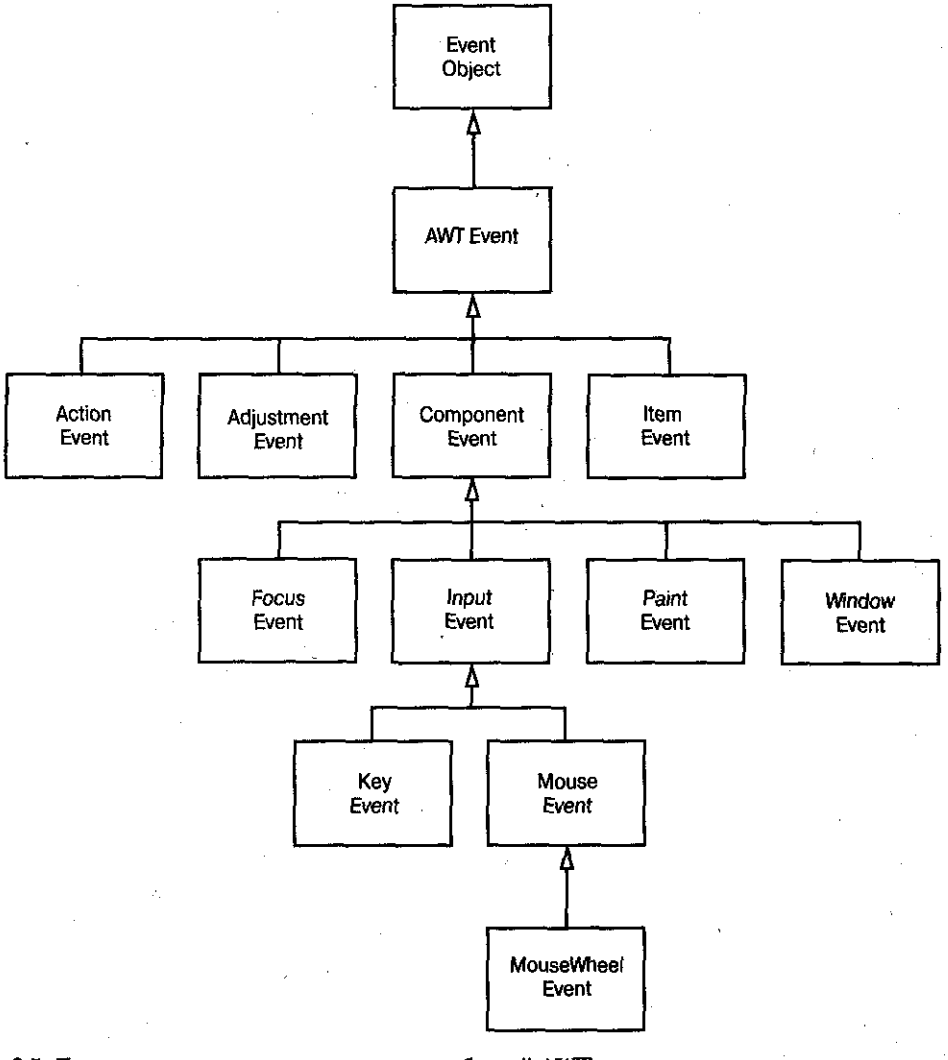


Рис. 8.5. Диаграмма наследования классов событий AWT

Некоторые компоненты пакета Swing генерируют другие объекты событий и непосредственно расширяют класс `EventObject`.

Объекты событий инкапсулируют информацию, которую источник события передает обработчику. По мере необходимости объекты событий можно проанализировать с помощью методов `getSource()` и `getActionSource()`.

Некоторые из классов событий AWT не имеют практической ценности. Например, пакет AWT помещает объекты `PaintEvent` в очередь событий, однако эти объекты обработчику не передаются. Программисты должны сами переопределять метод `paintComponent()`, чтобы управлять перерисовкой. AWT также генерирует ряд событий, интересующих лишь системных программистов; они могут использоваться для поддержки иероглифических языков, автоматизации тестирования роботов и выполнения других самых разнообразных задач. В этой книге мы не будем обсуждать специализированные типы событий, а также обойдем вниманием события, связанные с устаревшими и не используемыми сейчас компонентами AWT.

Ниже перечислены часто применяемые события AWT.

```
ActionEvent                KeyEvent
AdjustmentEvent            MouseEvent
FocusEvent                 MouseWheelEvent
ItemEvent                  WindowEvent
```

В этой и последующих главах будут приведены примеры, иллюстрирующие использование всех указанных типов событий.

Пакет `javax.swing.event` содержит дополнительные события, характерные для компонентов пакета Swing. Некоторые из них будут описаны в следующей главе.

Интерфейсы, используемые для создания обработчиков событий, имеют следующие имена.

```
ActionListener             MouseMotionListener
AdjustmentListener         MouseWheelListener
FocusListener              WindowListener
ItemListener               WindowFocusListener
KeyListener                WindowStateListener
MouseListener
```

С интерфейсами `ActionListener` и `WindowListener` вы уже знакомы. Они использовались в рассмотренных выше примерах.

Несмотря на то что пакет `javax.swing.event` содержит очень много интерфейсов, специфичных для компонентов из пакета Swing, в нем все еще используются основные интерфейсы из библиотеки AWT. В частности, они применяются для обработки событий общего характера.

Семь интерфейсов из пакета AWT, имеющих несколько методов, сопровождаются классами-адаптерами, реализующими пустые методы. (Оставшиеся интерфейсы содержат только по одному методу, поэтому для них классы-адаптеры не нужны.) Ниже перечислены имена классов-адаптеров.

```
FocusAdapter               MouseMotionAdapter
KeyAdapter                 WindowAdapter
MouseAdapter
```

В каждой прикладной программе используется много классов и интерфейсов, заслуживающих внимания. Класс, предназначенный для получения информации о событии, должен реализовывать интерфейс обработчика. Объект этого класса связывается с источником события. При возникновении события оно передается объекту, где обрабатывается с помощью методов, определенных в интерфейсе обработчика.

Программисты, работавшие на языках C и C++, могут поинтересоваться: зачем для обработки событий нужны все эти хитросплетения объектов, методов и интерфейсов? Ведь для создания графического пользовательского интерфейса на языке C++ можно использовать обратные вызовы с обобщенными указателями или обработчиками. Однако в языке Java этот механизм не работает. Модель обработки событий в языке Java строго типизирована: компилятор следит, чтобы события передавались только объектам, предназначенным для их обработки.

Семантические и низкоуровневые события в библиотеке AWT

В AWT события разделены на *низкоуровневые* (low-level) и *семантические* (semantic). Семантические события описывают действия пользователя, например щелчок на кнопке; следовательно, событие `ActionEvent` является семантическим. Низкоуровневые события обеспечивают возможность таких действий. Если пользователь щелкнул на кнопке, значит, он нажал кнопку мыши, возможно, переместил курсор по экрану и отпустил кнопку мыши (причем курсор мыши должен находиться в пределах кнопки). Семантические события могут быть также сгенерированы путем нажатия клавиш, например для перемещения по кнопкам с помощью клавиши <Tab>. Аналогично, действие с полосой прокрутки относится к семантическим событиям, в то время как перетаскивание объекта с помощью мыши относится к низкоуровневым.

Из классов пакета `java.awt.event`, описывающих семантические события, наиболее часто используются следующие.

- `ActionEvent` (щелчок на кнопке, выбор пункта меню, выбор пункта в списке, нажатие клавиши <Enter> при работе с полм редактирования).
- `AdjustmentEvent` (перемещение ползунка на полосе прокрутки).
- `ItemEvent` (выбор положения переключателя опций или пункта в списке).

Из низкоуровневых событий наиболее часто встречаются перечисленные ниже.

- `KeyEvent` (нажатие или отпускание клавиши).
- `MouseEvent` (нажатие, отпускание кнопки мыши, перемещение курсора мыши, перетаскивание курсора, т.е. его перемещение при нажатой кнопке).
- `MouseWheelEvent` (вращение колесика мыши).
- `FocusEvent` (получение или потеря фокуса ввода.).
- `WindowEvent` (изменение состояния окна).

В табл. 8.1 приведены наиболее важные интерфейсы обработчиков, события и источники событий.

Таблица 8.1. Объекты и интерфейсы, участвующие в обработке событий

Интерфейс	Методы	Параметр/ Методы доступа	Источник события
ActionListener	actionPerformed	ActionEvent • getActionCommand • getModifiers	AbstractButton JComboBox JTextField Timer
AdjustmentListener	adjustmentValue-Changed	AdjustmentEvent • getAdjustable • getAdjustmentType • getValue	JScrollbar
ItemListener	itemStateChanged	ItemEvent • getItem • getItemSelectable • getStateChange	AbstractButton JComboBox
FocusListener	focusGained focusLost	FocusEvent • isTemporary	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent • getKeyChar • getKeyCode • getKeyModifiersText • getKeyText • isActionKey	Component
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent • getClickCount • getX • getY • getPoint • translatePoint	Component
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
MouseWheelListener	mouseWheelMoved	MouseWheelEvent • getWheelRotation • getScrollAmount	Component
WindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated windowDeactivated	WindowEvent • getWindow	Window
WindowFocusListener	windowGainedFocus windowLostFocus	WindowEvent • getOppositeWindow	Window
WindowStateListener	windowStateChanged	WindowEvent • getOldState • getNewState	Window

Взаимосвязь основных понятий, используемых при обработке событий

Рассмотрим еще раз механизм делегирования событий, чтобы убедиться, что вы правильно понимаете отношения между классами событий, интерфейсами обработчиков и классами-адаптерами.

Источники событий — это компоненты пользовательского интерфейса, окна и меню. Операционная система извещает источник события о поведении интересующих его объектов, например о перемещении мыши или нажатии клавиши. Источник события описывает природу события в *объекте* события. Кроме того, он хранит набор *обработчиков* — объектов, которые следует вызвать при наступлении определенного события (рис. 8.6). Затем источник события вызывает соответствующий метод, объявленный в *интерфейсе обработчика*, для того, чтобы доставить информацию о событии требуемому объекту. С этой целью источник передает объект события методу обработчика. Обработчик анализирует объект события, извлекая из него информацию о выполненном действии. Например, чтобы обнаружить источник события, можно вызвать метод `getSource()`, а для того чтобы определить местонахождение курсора мыши — методы `getX()` и `getY()`.

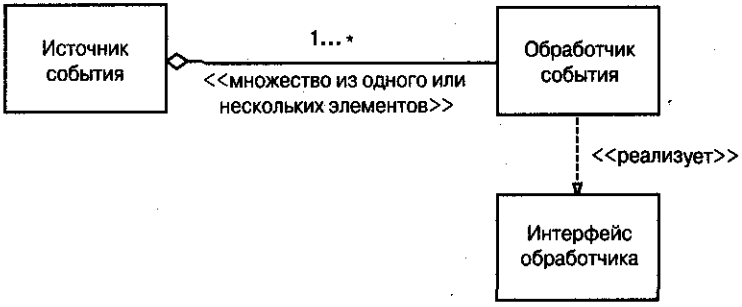


Рис. 8.6. Взаимосвязь между источниками событий и обработчиками

Обратите внимание на то, что интерфейсы `MouseListener` и `MouseMotionListener` отличаются друг от друга. Это сделано для повышения эффективности — есть много событий, связанных с мышью, которые не требуют реакции, например бесцельное перемещение курсора мыши по экрану. Поэтому часто обработчик должен заботиться лишь о нажатии кнопки мыши и не обращать внимания на ее произвольные перемещения.

Все низкоуровневые события являются потомками класса `ComponentEvent`. Этот класс содержит метод `getComponent()`, сообщающий о компоненте, породившем данное событие. Этот метод можно использовать вместо метода `getSource()`. Он возвращает то же значение, что и метод `getSource()`, но при этом возвращаемое значение приводится к типу `Component`. Например, если при вводе текста произойдет событие, связанное с нажатием клавиши, метод `getComponent()` вернет ссылку на соответствующее поле ввода.

API

- `Component getComponent()`

  Возвращает ссылку на компонент, являющийся источником события. Точно такая же ссылка возвращается при выполнении выражения `(Component)getSources()`.

Типы низкоуровневых событий

В этом разделе мы детально обсудим события, не связанные с конкретными компонентами пользовательского интерфейса, в частности, события, которые возникают при работе с клавиатурой и мышью. Подробный анализ семантических событий, порождаемых компонентами графического пользовательского интерфейса, проводится в следующей главе.

События, связанные с клавиатурой

Нажимая клавишу, пользователь порождает событие `KEY_PRESSED`, а отпуская ее — `KEY_RELEASED`. Оба этих события принадлежат классу `KeyEvent`. Их можно перехватить с помощью методов `keyPressed()` и `keyReleased()` любого класса, реализующего интерфейс `KeyListener`. В частности, эти методы можно использовать для отслеживания ошибочно нажатых клавиш. Третий метод, `keyTyped`, представляет собой сочетание двух предыдущих: он сообщает, какой *символ* соответствует нажатой клавише.

Перед тем как переходить к рассмотрению конкретного примера, уточним терминологию. В языке Java существует различие между *символами* (symbols) и виртуальными кодами клавиш (*virtual key codes*). Виртуальные коды клавиш обозначаются с помощью префикса `VK_`, например `VK_A` или `VK_SHIFT`. Они соответствуют клавишам клавиатуры; так, код `VK_A` означает клавишу <A>. Виртуальный код соответствует клавише, прописные и строчные буквы не различаются.

Виртуальные коды клавиш соответствуют их сканкодам.

Предположим, что пользователь ввел прописную букву "A", нажав клавишу <Shift> и клавишу <A>. При этом происходит *пять* событий.

1. Нажатие клавиши <Shift> (вызывается метод `keyPressed()` с параметром `VK_SHIFT`).
2. Нажатие клавиши <A> (вызывается метод `keyPressed()` с параметром `VK_A`).
3. Ввод буквы "A" (вызывается метод `keyTyped()` с параметром "A").
4. Отпускание клавиши <A> (вызывается метод `keyReleased()` с параметром `VK_A`).
5. Отпускание клавиши <Shift> (вызывается метод `keyReleased()` с параметром `VK_SHIFT`).

Если же пользователь вводит строчную букву "a", нажимая клавишу <A>, произойдут только три события.

1. Нажатие клавиши <A> (вызывается метод `keyPressed()` с параметром `VK_A`).
2. Ввод буквы "a" (вызывается метод `keyTyped()` с параметром "a").
3. Отпускание клавиши <A> (вызывается метод `keyReleased()` с параметром `VK_A`).

Итак, процедура `keyTyped()` сообщает, какой символ был введен ("A" или "a"), в то время как методы `keyPressed()` и `keyReleased()` сообщают, какие *клавиши* были нажаты.

При работе с методами keyPressed и keyReleased нужно сначала проверить код клавиши.

```
public void keyPressed(KeyEvent event)
{
   int keyCode = event.getKeyCode();
   ...
}
```

Код клавиши равен одной из следующих констант (их имена вполне очевидны). Эти константы определены в классе KeyEvent.

```
VK_A . . . VK_Z
VK_0 . . . VK_9
VK_COMMA, VK_PERIOD, VK_SLASH, VK_SEMICOLON, VK_EQUALS
VK_OPEN_BRACKET, VK_BACK_SLASH, VK_CLOSE_BRACKET
VK_BACK_QUOTE, VK_QUOTE
VK_GREATER, VK_LESS, VK_UNDERSCORE, VK_MINUS
VK_AMPERSAND, VK_ASTERISK, VK_AT, VK_BRACELEFT, VK_BRACERIGHT
VK_LEFT_PARENTHESIS, VK_RIGHT_PARENTHESIS
VK_CIRCUMFLEX, VK_COLON, VK_NUMBER_SIGN, VK_QUOTEDBL
VK_EXCLAMATION_MARK, VK_INVERTED_EXCLAMATION_MARK
VK_DEAD_ABOVEDOT, VK_DEAD_ABOVERING, VK_DEAD_ACUTE
VK_DEAD_BREVE
VK_DEAD_CARON, VK_DEAD_CEDILLA, VK_DEAD_CIRCUMFLEX
VK_DEAD_DIAERESIS
VK_DEAD_DOUBLEACUTE, VK_DEAD_GRAVE, VK_DEAD_IOTA, VK_DEAD_MACRON
VK_DEAD_OGONEK, VK_DEAD_SEMIVOICED_SOUND, VK_DEAD_TILDE,
VK_DEAD_VOICED_SOUND
VK_DOLLAR, VK_EURO_SIGN
VK_SPACE, VK_ENTER, VK_BACK_SPACE, VK_TAB, VK_ESCAPE
VK_SHIFT, VK_CONTROL, VK_ALT, VK_ALT_GRAPH, VK_META
VK_NUM_LOCK, VK_SCROLL_LOCK, VK_CAPS_LOCK
VK_PAUSE, VK_PRINTSCREEN
VK_PAGE_UP, VK_PAGE_DOWN, VK_END, VK_HOME, VK_LEFT, VK_UP, VK_RIGHT,
VK_DOWN
VK_F1 . . .VK_F24
VK_NUMPAD0 . . . VK_NUMPAD9
VK_KP_DOWN, VK_KP_LEFT, VK_KP_RIGHT, VK_KP_UP
VK_MULTIPLY, VK_ADD, VK_SEPARATER [sic], VK_SUBTRACT, VK_DECIMAL,
VK_DIVIDE
VK_DELETE, VK_INSERT
VK_HELP, VK_CANCEL, VK_CLEAR, VK_FINAL
VK_CONVERT, VK_NONCONVERT, VK_ACCEPT, VK_MODECHANGE
VK_AGAIN, VK_ALPHANUMERIC, VK_CODE_INPUT, VK_COMPOSE, VK_PROPS
VK_STOP
VK_ALL_CANDIDATES, VK_PREVIOUS_CANDIDATE
VK_COPY, VK_CUT, VK_PASTE, VK_UNDO
VK_FULL_WIDTH, VK_HALF_WIDTH
VK_HIRAGANA, VK_KATAKANA, VK_ROMAN_CHARACTERS
VK_KANA, VK_KANJI
VK_JAPANESE_HIRAGANA, VK_JAPANESE_KATAKANA, VK_JAPANESE_ROMAN
VK_WINDOWS, VK_CONTEXT_MENU
VK_UNDEFINED
```

Чтобы определить текущее состояние клавиш <Shift>, <Control>, <Alt> и <Meta>, можно, конечно, отслеживать коды VK_SHIFT, VK_CONTROL и VK_META, но это очень сложно. Достаточно использовать методы isShiftDown(), isControlDown(), isAltDown() и isMetaDown().

Например, следующий фрагмент кода проверяет, нажимается ли клавиша со стрелкой вправо при нажатой клавише <Shift>.

```
public void keyPressed(KeyEvent event)
{
   int keyCode == event.getKeyCode();
   if (keyCode == keyEvent.VK_RIGTH && event.isShiftDown())
   {
      ...
   }
}
```

В методе keyTyped() вызывается метод getKeyChar(), позволяющий определить, какой символ был введен.

Метод keyTyped() можно применять не для всех клавиш. Он предназначен лишь для отслеживания таких клавиш, которые генерируют символ в кодировке Unicode. Чтобы проверять действия с управляющими клавишами, например вызывающими перемещение курсора, надо использовать метод keyPressed().

Программа, представленная в листинге 8.3, иллюстрирует обработку нажатия клавиш. Результат ее работы показан на рис. 8.7. Программа представляет собой простую реализацию игрушки Etch-A-Sketch™ и позволяет создавать простые эскизы.

Перо можно перемещать вверх, вниз, влево и вправо, пользуясь клавишами со стрелками. Если при этом нажать клавишу <Shift>, перо переместится на большее расстояние. Тот, кто имеет опыт работы с редактором vi, может переназначить клавиши и для перемещения пера по экрану использовать буквы нижнего регистра: "h", "j", "k" и "l". Те же буквы, но верхнего регистра ("H", "J", "K" и "L"), перемещают перо на большее расстояние. Отслеживание состояния клавиш осуществляется с помощью метода keyPressed(), а получение символов — посредством метода keyTyped().

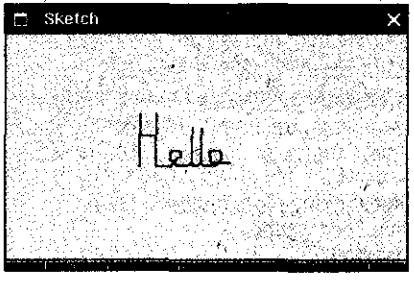


Рис. 8.7. Программа рисования

В данной программе есть одна деталь: обычно панель не может получать никаких событий, связанных с нажатием клавиш. Для того чтобы обойти это ограничение, мы вызываем метод setFocusable(). Вопросы получения фокуса ввода мы рассмотрим далее в этой главе.

Листинг 8.3. Содержимое файла Sketch.java

```
import java.awt.*;
import java.awt.geom.*;
import java.util.*;
import java.awt.event.*;
import javax.swing.*;

public class Sketch
{
   public static void main(String[] args)
   {
      SketchFrame frame = new SketchFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с панелью для рисования фигур.
*/
class SketchFrame extends JFrame
{
   public SketchFrame()
   {
      setTitle("Sketch");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление панели к фрейму.

      SketchPanel panel = new SketchPanel();
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
   Панель для рисования с помощью клавиатуры.
*/
class SketchPanel extends JPanel
{
   public SketchPanel()
   {
      last = new Point2D.Double(100, 100);
      lines = new ArrayList<Line2D>();
      KeyHandler listener = new KeyHandler();
      addKeyListener(listener);
      setFocusable(true);
   }

   /**
      Добавление нового отрезка к рисунку.
      @param dx Перемещение в направлении x
      @param dy Перемещение в направлении y
```

```
   */
   public void add(int dx, int dy)
   {
      // Вычисление конца нового отрезка.
      Point2D end = new Point2D.Double(last.getX() + dx,
         last.getY() + dy);

      // Добавление нового отрезка.
      Line2D line = new Line2D.Double(last, end);
      lines.add(line);
      repaint();

      // Запоминается очередная конечная точка.
      last = end;
   }

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;

      // Рисование всех линий.
      for (Line2D l : lines)
         g2.draw(l);
   }

   private Point2D last;
   private ArrayList<Line2D> lines;

   private static final int SMALL_INCREMENT = 1;
   private static final int LARGE_INCREMENT = 5;

   private class KeyHandler implements KeyListener
   {
      public void keyPressed(KeyEvent event)
      {
         int keyCode = event.getKeyCode();

         // Установка расстояния.
         int d;
         if (event.isShiftDown())
            d = LARGE_INCREMENT;
         else
            d = SMALL_INCREMENT;

         // Добавление нового отрезка.
         if (keyCode == KeyEvent.VK_LEFT) add(-d, 0);
         else if (keyCode == KeyEvent.VK_RIGHT) add(d, 0);
         else if (keyCode == KeyEvent.VK_UP) add(0, -d);
         else if (keyCode == KeyEvent.VK_DOWN) add(0, d);
      }

      public void keyReleased(KeyEvent event) {}

      public void keyTyped(KeyEvent event)
      {
```

```
            char keyChar = event.getKeyChar();

            // Установка расстояния.
            int d;
            if (Character.isUpperCase(keyChar))
            {
               d = LARGE_INCREMENT;
               keyChar = Character.toLowerCase(keyChar);
            }
            else
               d = SMALL_INCREMENT;

            // Добавление нового отрезка.
            if (keyChar == 'h') add(-d, 0);
            else if (keyChar == 'l') add(d, 0);
            else if (keyChar == 'k') add(0, -d);
            else if (keyChar == 'j') add(0, d);
         }
      }
}
```

API **java.awt.event.KeyEvent 1.1**

- `char getKeyChar()`

  Возвращает символ, введенный пользователем.

- `int getKeyCode()`

  Возвращает виртуальный код клавиши, соответствующий данному событию.

- `boolean isActionKey()`

  Возвращает значение `true`, если данное событие связано с одной из следующих клавиш: <Home>, <End>, <Page Up>, <Page Down>, <Up>, <Down>, <Left>, <Right>, <F1 ... F24>, <Print Screen>, <Scroll Lock>, <Caps Lock>, <Num Lock>, <Pause>, <Insert>, <Delete>, <Enter>, <Backspace> и <Tab>.

- `static String getKeyText(int keyCode)`

  Возвращает строку, описывающую клавишу. Например, вызов `getKetText(KeyEvent.VK_END)` возвращает строку `"END"`.

- `static String getKeyModifiersText(int modifiers)`

  Возвращает строку, описывающую модификаторы, например <Shift> или <Ctrl>+<Shift>.

  Параметр `modifiers` Состояние модификаторов

API **java.awt.event.InputEvent 1.1**

- `int getModifiers()`

  Возвращает целое число, описывающее состояние модификаторов <Shift>, <Ctrl> <Alt> и <Meta>. Этот метод применяется как для клавиатуры, так и для мыши. Для того чтобы проверить, установлен ли определенный разряд, при-

меняются маски SHIFT_MASK, CTRL_MASK, META_MASK или вызывается один из перечисленных ниже методов.

- `boolean isShiftDown()`
- `boolean isControlDown()`
- `boolean isAltDown()`
- `boolean isAltGraphDown()` **1.2**
- `boolean isMetaDown()`

Эти методы возвращают значение `true`, если клафиша-модификатор в момент генерации данного события была нажата.

События, связанные с мышью

Если пользователь должен только щелкать на кнопках или выбирать пункты меню, обрабатывать явным образом события, связанные с мышью, не нужно. Эти операции автоматически поддерживаются компонентами пользовательского интерфейса, а затем преобразовываются в соответствующие семантические события. Однако, если пользователь должен иметь возможность рисовать с помощью мыши, необходимо отслеживать ее перемещения, щелчки и события, связанные с перетаскиванием объектов.

В данном разделе мы рассмотрим простой графический редактор, позволяющий создавать, перемещать и стирать квадраты на холсте (рис. 8.8).

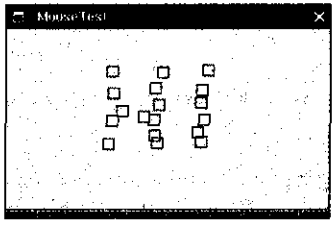


Рис. 8.8. Программа, демонстрирующая обработку событий, связанных с мышью

Когда пользователь щелкает мышью, вызываются три метода объекта-обработчика: `mousePressed()`, если кнопка мыши была нажата, `mouseReleased()`, если кнопка была отпущена, и `mouseClicked`. Если надо отследить только сам щелчок, первые два метода использовать не обязательно. Пользуясь методами `getX()` и `getY()` объекта `MouseEvent`, передаваемого в качестве параметра, можно определить координаты курсора мыши в момент щелчка. Если нужно различать обычный, двойной и тройной (!) щелчки, используется метод `getClickCount()`.

Некоторые разработчики так реализуют свое программное обеспечение, что для управления интерфейсными элементами пользователю приходится щелкать мышью при нажатых клавишах. В исходной библиотеке API маски, соответствующие двум кнопкам мыши, совпадали с масками клавиш-модификаторов:

```
BUTTON2_MASK == ALT_MASK
BUTTON3_MASK == META_MASK
```

Вероятно, это было сделано для того, чтобы пользователи однокнопочных мышей могли имитировать нажатие второй кнопки с помощью нажатия клавиши на клавиа-

туре. Однако в пакете JDK 1.4 использован другой подход. Теперь в нем предусмотрено несколько масок.

```
BUTTON1_DOWN_MASK
BUTTON2_DOWN_MASK
BUTTON3_DOWN_MASK
SHIFT_DOWN_MASK
CTRL_DOWN_MASK
ALT_DOWN_MASK
ALT_GRAPH_DOWN_MASK
META_DOWN_MASK
```

Метод `getModifiersEx()` позволяет распознавать события, связанные с нажатием кнопок мыши при нажатых клавишах.

Обратите внимание на то, что маска `BUTTON3_DOWN_MASK` позволяет проверять нажатие правой кнопки мыши в среде Windows. Например, для определения, нажата ли правая клавиша, можно воспользоваться следующим кодом:

```
if ((event.getModifiersEx() & InputEvent.BUTTON3_DOWN_MASK) != 0)
  // Код, обрабатывающий нажатие правой кнопки
```

В программе, которая будет рассмотрена ниже, используются методы `mousePressed()` и `mouseClicked()`. Если щелкнуть кнопкой мыши на пикселе, не принадлежащем ни одному нарисованному квадрату, на экране появится новый квадрат. Эта процедура реализована в методе `mousePressed()`, поэтому компьютер отреагирует на щелчок мыши немедленно, не дожидаясь освобождения кнопки мыши. Двойной щелчок внутри какого-нибудь квадрата приведет к тому, что он будет стерт. Эта процедура реализована в методе `mouseClicked()`, поскольку щелчки мыши нужно подсчитывать.

```
public void mousePressed(MouseEvent event)
{
   current = find(event.getPoint());
   if (current == null) // Вне квадрата.
      add(event.getPoint());
}

public void mouseClicked(MouseEvent event)
{
   current = find(event.getPoint());
   if (current != null && event.getClickCount() >= 2)
      remove(current);
}
```

При перемещении мыши по окну последнее получает постоянный поток событий, связанных с движением мыши. В большинстве приложений эти события игнорируются. Однако в нашем примере программа отслеживает события, чтобы изменить вид курсора (на крестообразный), как только он выйдет за пределы квадрата. Эта процедура реализована в методе `getPredefinedCursor()` в классе `Cursor`. В табл. 8.2 приведены константы, используемые в этом методе, и показаны виды курсоров, предусмотренные в среде Windows.

Таблица 8.2. Виды курсоров

Пиктограмма	Константа
	DEFAULT_CURSOR
	CROSSHAIR_CURSOR
	HAND_CURSOR
	MOVE_CURSOR
	TEXT_CURSOR
	WAIT_CURSOR
	N_RESIZE_CURSOR
	NE_RESIZE_CURSOR
	E_RESIZE_CURSOR
	SE_RESIZE_CURSOR
	S_RESIZE_CURSOR
	SW_RESIZE_CURSOR
	W_RESIZE_CURSOR
	NW_RESIZE_CURSOR

(Обратите внимание на то, что некоторые виды курсоров выглядят одинаково. На других платформах они, возможно, будут различаться.)

Изображения курсоров можно найти в каталоге `jre/lib/images/cursors`. *В файле* `cursor.properties` определены точки, являющиеся указателями. Например, если курсор имеет вид руки, то указателем считается кончик указательного пальца. Если *курсор имеет вид лупы, то указателем является центр этой лупы.*

Рассмотрим метод mouseMoved(), объявленный в интерфейсе MouseMotionListener и реализованный в нашей программе.

```
public void mouseMoved(MouseEvent event)
{
   if (find(event.getPoint()) == null)
      setCursor(Cursor.getDefaultCursor());
   else
      setCursor(Cursor.getPredefinedCursor
         (Cursor.CROSSHAIR_CURSOR));
}
```

Используя метод createCustomCursor() из класса Toolkit, можно определить новую форму курсора.

```
Toolkit tk = Toolkit.getDefaultToolkit();
Image img = tk.getImage("dynamite.gif");
Cursor dynamiteCursor = tk.createCustomCursor(img,
   new Point(10, 10), "dynamite stick");
```

Первый параметр метода createCustomCursor() задает изображение курсора. Второй параметр определяет координаты точки, используемой в качестве указателя. Третий параметр — строка, описывающая курсор. Эту строку можно использовать для вспомогательных целей, например, сообщить голосом вид курсора. Такая возможность важна при адаптации программ для пользователей с нарушениями зрения.

Если перемещение мыши осуществляется при нажатой кнопке, вместо mouseClicked() вызывается метод mouseDragged(). В нашем примере квадраты можно перетаскивать по экрану. При этом квадрат перемещается так, чтобы его центр располагался в той точке, на которую указывает мышь. Содержимое холста перерисовывается.

```
public void mouseDragged(MouseEvent event)
{
   if (cursor >= null)
   {
      int x = event.getX();
      int y = event.getY();

      current.setFrame(
        x - SIDELENGTH / 2,
        y - SIDELENGTH / 2,
        SIDELENGTH,
        SIDELENGTH);
      repaint();
   }
}
```

Метод mouseMoved() вызывается, только если указатель мыши находится внутри компонента. Однако метод mouseDragged() вызывается даже тогда, когда указатель мыши пребывает вне компонента.

Есть еще два метода, обрабатывающих события, связанные с мышью: mouseEntered() и mouseExited(). Эти методы вызываются, когда указатель мыши входит в пределы компонента и выходит из него.

Заметим еще одну особенность, связанную с обработкой событий мыши. В ответ на щелчок мышью вызывается метод mouseClicked(), являющийся частью интерфейса MouseListener. Поскольку во многих приложениях отслеживаются только щелчки мышью, а также потому, что перемещения мыши происходят слишком часто, события, связанные с перемещением мыши и перетаскиванием, определяются в отдельном интерфейсе MouseMotionListener.

В нашей программе нас интересуют оба типа событий, связанных с мышью. В ней определены два внутренних класса: MouseHandler и MouseMotionHandler. Класс MouseHandler расширяет класс MouseAdapter, поскольку он определяет только два *из пяти методов класса* MouseListener. *Класс* MouseMotionHandler *реализует интерфейс* MouseMotionListener и определяет оба его метода. Текст программы приведен в листинге 8.4.

Листинг 8.4. Содержимое файла MouseTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.awt.geom.*;
import javax.swing.*;

public class MouseTest
{
   public static void main(String[] args)
   {
      MouseFrame frame = new MouseFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм, содержащий панель, предназначенную для проверки
   операций с мышью.
*/
class MouseFrame extends JFrame
{
   public MouseFrame()
   {
      setTitle("MouseTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление панели к фрейму.

      MousePanel panel = new MousePanel();
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
```

```
}

/**
   Панель, поддерживающая добавление и удаление квадратов
   с помощью мыши.
*/
class MousePanel extends JPanel
{
   public MousePanel()
   {
      squares = new ArrayList<Rectangle2D>();
      current = null;

      addMouseListener(new MouseHandler());
      addMouseMotionListener(new MouseMotionHandler());
   }

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;

      // Рисование всех квадратов.
      for (Rectangle2D r : squares)
         g2.draw(r);
   }

   /**
     Нахождение первого квадрата, который содержит заданную точку.
     @param p Точка
     @return Первый квадрат, содержащий точку p
   */
   public Rectangle2D find(Point2D p)
   {
      for (Rectangle2D r : squares)
      {
         if (r.contains(p)) return r;
      }
      return null;
   }

   /**
      Добавление квадрата к набору.
      @param p Центр квадрата
   */
   public void add(Point2D p)
   {
      double x = p.getX();
      double y = p.getY();

      current = new Rectangle2D.Double(
         x - SIDELENGTH / 2,
         y - SIDELENGTH / 2,
         SIDELENGTH,
         SIDELENGTH);
      squares.add(current);
```

```
      repaint();
   }

   /**
      Удаление квадрата из набора.
      @param s Удаляемый квадрат
   */
   public void remove(Rectangle2D s)
   {
      if (s == null) return;
      if (s == current) current = null;
      squares.remove(s);
      repaint();
   }

   private static final int SIDELENGTH = 10;
   private ArrayList<Rectangle2D> squares;
   private Rectangle2D current;
   // Квадрат, в пределах которого находится курсор.

   private class MouseHandler extends MouseAdapter
   {
      public void mousePressed(MouseEvent event)
      {
         // Если курсор находится за пределами всех квадратов,
         // добавляется новый квадрат
         current = find(event.getPoint());
         if (current == null)
            add(event.getPoint());
      }

      public void mouseClicked(MouseEvent event)
      {
         // После двойного щелчка текущий квадрат удаляется.
         current = find(event.getPoint());
         if (current != null && event.getClickCount() >= 2)
            remove(current);
      }
   }

   private class MouseMotionHandler
      implements MouseMotionListener
   {
      public void mouseMoved(MouseEvent event)
      {
         // Если курсор находится в пределах квадрата,
         // он принимает крестообразный вид.

         if (find(event.getPoint()) == null)
            setCursor(Cursor.getDefaultCursor());
         else
setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
      }
```

```
      public void mouseDragged(MouseEvent event)
      {
         if (current != null)
         {
            int x = event.getX();
            int y = event.getY();

            // Перетаскивание текущего квадрата в точку (x, y)
            current.setFrame(
               x - SIDELENGTH / 2,
               y - SIDELENGTH / 2,
               SIDELENGTH,
               SIDELENGTH);
            repaint();
         }
      }
   }
}
```

java.awt.event.MouseEvent 1.1

- `int getX()`
- `int getY()`
- `Point getPoint()`

  Возвращает горизонтальную (x) и вертикальную (y) координаты или точку, соответствующую событию. Отсчет производится от левого верхнего угла компонента, являющегося источником события.

- `void translatePoint(int x, int y)`

  Преобразует координаты события, перемещая точку на x единиц горизонтально и y единиц вертикально.

- `int getClickCount()`

  Возвращает количество последовательных щелчков мыши, связанных с данным событием. (Интервал времени, в пределах которого подсчитываются щелчки, зависит от операционной системы.)

java.awt.event.InputEvent 1.

- `int getModifiersEx()` **1.4**

  Возвращает модификаторы для события. При проверке возвращаемого значения используются перечисленные ниже маски.

  ```
  BUTTON1_DOWN_MASK
  BUTTON2_DOWN_MASK
  BUTTON3_DOWN_MASK
  SHIFT_DOWN_MASK
  CTRL_DOWN_MASK
  ALT_DOWN_MASK
  ALT_GRAPH_DOWN_MASK
  META_DOWN_MASK
  ```

- `static int getModifiersExText(int modifiers)` **1.4**

  Возвращает строку, описывающую модификаторы, например `Shift+Button1`.

java.awt.Toolkit 1.0

- `public Cursor createCustomCursor(Image image, Point hotSpot, String name)` **1.2**

  Создает новый объект курсора.

Параметры	`image`	*Изображение активного курсора*
	`hotSpot`	Точка, считающаяся указателем
	`name`	*Описание курсора*

java.awt.Component 1.0

- `public void setCursor(Cursor cursor)` **1.1**

  Изменяет внешний вид курсора.

События, связанные с фокусом ввода

Пользуясь мышью, можно указать любой объект на экране. Однако при вводе текста нажатие клавиши должно передаваться конкретному объекту на экране. *Оконный диспетчер* направляет все нажатия клавиш *активному окну* (active window). Обычно активное окно отличается от остальных цветом строки заголовка. Активным может быть лишь одно окно.

Предположим, что активным окном управляет программа, написанная на языке Java. Это окно получает информацию о нажатии клавиш и направляет ее конкретному *компоненту*. В таком случае говорят, что этот компонент обладает фокусом ввода. Аналогично тому, как активное окно выделяется среди остальных цветом строки заголовка, компонент, имеющий фокус, визуально отличается от других. В поле ввода текста мигает курсор, активная кнопка выделяется прямоугольником, обрамляющим ее заголовок, и т.д. Если фокус принадлежит полю редактирования, в нем можно вводить текст. Если фокусом обладает кнопка, на ней можно щелкнуть, нажав клавишу пробела.

В каждый момент времени фокус ввода может принадлежать только одному компоненту. Компонент может *потерять фокус ввода*, если пользователь выбрал другой компонент. Этому компоненту передается фокус ввода. Компонент получает фокус, если пользователь щелкнул на нем мышью. Для этой цели можно воспользоваться также клавишей <Tab>, которая перемещает фокус с одного компонента на другой. Таким образом, можно пройтись по всем компонентам, которые потенциально могут обладать фокусом. По умолчанию перемещение по компонентам библиотеки Swing осуществляется слева направо, а затем сверху вниз в соответствии с их расположением в контейнере. Порядок обхода компонентов можно изменить (подробная информация об этом приведена в следующей главе).

К счастью, большинство прикладных программистов не слишком беспокоятся об обработке событий, связанных с фокусом. До появления пакета JDK 1.4 этот механизм обычно использовался для проверки возможных ошибок или верификации данных. Допустим, что в текстовое поле вводится номер кредитной карточки и после завершения редактирования пользователь переходит в новое поле. В этот момент можно

перехватить событие, связанное с потерей фокуса. Если номер кредитной карточки не соответствует ожидаемому формату, можно вывести сообщение об ошибке и вернуть фокус предыдущему полю ввода. Однако в пакете JDK 1.4 есть более простые и надежные механизмы верификации. Мы обсудим их в главе 9.

К сожалению, в ранних версиях пакета JDK обработка событий, связанных с фокусом ввода, была реализована не слишком удачно. Пользователи обнаружили больше сотни разных ошибок. Это явление объясняется двумя причинами. Во-первых, фокус компонента взаимодействует с фокусом окна, который находится в компетенции оконного диспетчера. Следовательно, поведение фокуса зависит от платформы, на которой выполняется приложение. Во-вторых, реализация кода, очевидно, быстро вышла из-под контроля, особенно в сочетании с неудовлетворительной реализацией механизма отслеживания фокуса в пакете JDK 1.3.

Уинстон Черчилль (Winston Churchill) однажды сказал: "Американцы всегда поступают правильно... после того как исчерпают все остальные возможности". Очевидно, то же самое относится и к группе разработчиков языка Java. Они все же реализовали правильный механизм отслеживания фокусов в пакете JDK 1.4. В этом пакете полностью переработан код обработки фокуса и подробно описано его ожидаемое поведение, включая неизбежную зависимость от платформы.

Спецификацию механизма обработки фокуса ввода можно найти на Web-странице `http://java.sun.com/j2se/~1.4/docs/api/java/awt/doc-files/FocusSpec.html`.

Некоторые компоненты, например метки и панели, по умолчанию не получают фокус ввода, поскольку предполагается, что они используются для отображения сообщений или объединения нескольких компонентов в одно целое. Если вы хотите написать приложение, которое позволяло бы рисовать на панели, нажимая клавиши, то необходимо сделать так, чтобы панель могла обладать фокусом. Работая с JDK 1.4, для этого достаточно вызвать метод `setFocusable()`.

```
panel.setFocusable(true);
```

В старых версиях JDK переопределение метода `isFocusTraversable()`, принадлежащего компоненту, приводило к такому же результату. Однако в старых пакетах использовалась другая концепция фокуса ввода. Это различие вводило пользователей в заблуждение, поэтому метод `isFocusTraversable()` был объявлен как не рекомендованный к применению.

В оставшейся части раздела мы обсудим детали обработки событий, связанных с фокусом ввода. Этот текст вы можете спокойно пропустить, пока не почувствуете необходимость использовать описанный механизм в своих приложениях.

В пакете JDK 1.4 используются следующие понятия.

- *Владелец фокуса* (focus owner), т.е. компонент, обладающий фокусом ввода.
- Окно, содержащее владельца фокуса.
- *Активное окно* (active window), т.е фрейм или диалоговое окно, содержащее владельца фокуса.

Окно, содержащее владельца фокуса, обычно является активным. Однако возможна ситуация, когда владелец фокуса содержится в окне верхнего уровня, не являющемся фреймом, например во всплывающем меню.

Чтобы получить эту информацию о владельце, об окне, содержащем его, и активном окне, *сначала надо получить диспетчер фокуса клавиатуры.*

```
KeyboardFocusManager manager =
   KeyboardFocusManager.getCurrentKeyboardFocusManager();
```

Затем следует вызвать приведенные ниже методы.

```
Component owner = manager.getFocusOwner();
Window focused = manager.getFocusedWindow();
Window active = manager.getActiveWindow(); // Фрейм или диалоговое окно
```

Для регистрации перемещений фокуса ввода необходимо связать обработчики событий с компонентами или окнами. Обработчик должен реализовывать два метода: `focusGained()` и `focusLost()`. Эти методы вызываются тогда, когда *источник* события получает фокус ввода или теряет фокус. Каждому из этих методов передается параметр `FocusEvent`. В этом классе также есть два полезных метода: метод `getComponent()` сообщает, обладает ли компонент фокусом ввода, а метод `isTemporary()` возвращает значение `true`, если фокус ввода передан временно. Временная передача фокуса происходит тогда, когда компонент теряет управление, но впоследствии автоматически получает его обратно. Например, если пользователь переходит к другому окну, а затем возвращается к предыдущему, исходный компонент снова получает фокус ввода.

В JDK 1.4 реализован механизм доставки событий, связанных с фокусом ввода для окна. Для этого с окном связывается класс `WindowFocusListener` и переопределяются методы `windowGainedFocus()` и `windowLostFocus()`.

Кроме того, в пакете JDK 1.4 введено понятие "дополняющего", или "противоположного", компонента или окна. Когда компонент или окно теряют фокус ввода, "противоположный" компонент или окно получают фокус. И наоборот, когда исходный компонент или окно вновь получает фокус, их "противоположное" окно теряет его. Метод `getOppositeComponent()` класса `FocusEvent` возвращает "противоположный" компонент, а метод `getOppositeWindow()` класса `WindowEvent` – "противоположное" окно.

Фокус ввода можно запрограммировать с помощью метода `requestComponent()` класса `Component`. Однако поведение фокуса существенно зависит от платформы, на которой выполняется приложение. Для создания платформенно-независимых программ в пакете JDK 1.4 предусмотрен метод `requestFocusInWindow()` класса `Component`. Этот метод работает, только если компонент содержится внутри окна.

Не следует считать, что ваш компонент обладает фокусом ввода лишь на основании того, что метод `requestFocus()` или `requestFocusInWindow()` вернул значение `true`. Лучше дождаться события `FOCUS_GAINED`.

Некоторые программисты неверно понимают суть события `FOCUS_LOST` и пытаются предотвратить получение фокуса другим компонентом, вызывая обработчик `focusLost`. Однако к этому времени компонент уже потерял фокус ввода. Если вам

необходимо отслеживать перехват фокуса конкретным компонентом, необходимо инсталлировать "обработчик, обладающий правом вето" и запретить изменения свойства `focusOwner`. Подробно данный механизм описан в главе 8 тома 2.

java.awt.Component 1.0

- `void requestFocus()`

  Запрашивает передачу фокуса данному компоненту.

- `Boolean requestFocusInWindow()` **1.4**

  Запрашивает передачу фокуса данному компоненту. Возвращает значение `false`, если данный компонент не содержится в окне или запрос невозможно выполнить по другой причине. Возвращает значение `true`, если запрос можно выполнить.

- `void setFocusable(boolean b)` **1.4**
- `boolean isFocusable()` **1.4**

  Устанавливает или определяет способность компонента обладать фокусом ввода. Если параметр `b` имеет значение `true`, значит, компонент может получать фокус.

- `boolean isFocusOwner()` **1.4**

  Возвращает значение `true`, если компонент в данный момент обладает фокусом ввода.

java.awt.KeyboardFocusManager 1.

- `static KeyboardFocusManager getCurrentKeyboardFocusManager()`

  Определяет текущий диспетчер фокуса.

- `Component getFocusowner()`

  Определяет компонент, владеющий фокусом ввода, или возвращает ссылку `null`, если компонент, который обладает фокусом, не управляется диспетчером.

- `Window getFocusWindow()`

  Определяет окно, которое содержит компонент, владеющий фокусом ввода, или возвращает ссылку `null`, если компонент, который обладает фокусом, не управляется диспетчером.

- `Window getActiveWindow()`

  Определяет диалоговое окно или фрейм, содержащее окно, в котором находится компонент, владеющий фокусом ввода. Если окно не управляется диспетчером, возвращает ссылку `null`.

java.awt.Window 1.0

- `boolean isFocused()` **1.4**

  Возвращает значение `true`, если окно содержит элемент, обладающий фокусом ввода.

- `boolean isActive()` **1.4**

  Возвращает значение `true`, если фрейм или диалоговое окно является активным. Строка заголовка активного фрейма или диалогового окна обычно выделяется цветом.

API `java.awt.event.FocusEvent` **1.1**

- `Component getOppositeComponent()` **1.4**

  Возвращает компонент, который потерял фокус в обработчике `focusGained` или получил фокус в обработчике `focusLost`.

API `java.awt.event.WindowEvent` **1.4**

- `Component getOppositeWindow()` **1.4**

  Возвращает окно, потерявшее фокус в обработчике `windowGainedFocus`, либо окно, которое получило фокус в обработчике `windowLostFocus`, либо окно, активизированное в обработчике `windowActivated`, или окно, деактивизированное в обработчике `windowDeactivated`.

API `java.awt.event.WindowFocusListener` **1.4**

- `void windowGainedFocus(WindowEvent event)`

  Вызывается, когда окно, являющееся источником события, получает фокус ввода.

- `void windowLostFocus(windowEvent event)`

  Вызывается, когда окно, являющееся источником события, теряет фокус ввода.

Действия

Одну и ту же команду можно выполнить разными способами. Пользователь может выбрать пункт меню, нажать соответствующую клавишу или щелкнуть на кнопке панели инструментов. В этом случае очень удобна рассматриваемая нами модель событий: достаточно связать все эти события с одним и тем же обработчиком. Допустим, что `blueAction` — это объект некоего класса (скажем, `ColorAction`), реализующего интерфейс `ActionListener` и изменяющего цвет фона на синий. Можно связать один и тот же объект с разными источниками событий, перечисленными ниже.

- Кнопка Blue на панели инструментов.
- Пункт меню Blue.
- Нажатие клавиш <Ctrl+B>.

Команда, изменяющая цвет, выполняется одинаково, независимо от того, что привело к ее выполнению — щелчок на кнопке, выбор пункта меню или нажатие клавиш.

В пакете Swing предусмотрен очень полезный механизм, позволяющий инкапсулировать команды и связывать их с несколькими источниками событий. Этот механизм представляет собой интерфейс `Action`. *Действие* (action) — это объект, инкапсулирующий описание команды (в виде текстовой строки или пиктограммы) и параметры, необходимые для выполнения программы (например, нужный цвет).

Интерфейс Action содержит следующие методы:

```
Void actionPerformed(ActionEvent event)
void setEnabled(boolean b)
boolean isEnabled()
void pitValue(String key, Object value)
Object getValue(String key)
void addPropertyChangeListener(PropertyChangeListener listener)
void removePropertyChangeListener(PropertyChangeListener listener)
```

Первый метод похож на метод интерфейса ActionListener: действительно, интерфейс Action расширяет интерфейс ActionListener. Следовательно, вместо *объекта класса, реализующего интерфейс* ActionListener, *можно использовать* объект класса, реализующего интерфейс Action.

Следующие два метода позволяют блокировать и разблокировать действие, а также проверить, является ли указанное действие разблокированным (т.е. доступным) в данный момент. Если пункт меню или кнопка панели инструментов связаны с заблокированным действием, они выделяются светло-серым цветом.

Методы putValue() getValue() позволяют записывать и извлекать из памяти произвольные пары имя-значение объектов класса, реализующего интерфейс Action. Предопределенные строки, такие как Action.NAME и Action.SMALL_ICON, содержат имена и пиктограммы объектов действий.

```
Action.putValue(Action.NAME, "Blue");
action.putValue(Action.SMALL_ICON,
   new ImageIcon("blue-ball.gif"));
```

В табл. 8.3 приведены предопределенные имена.

Таблица 8.3. Заранее определенные имена действий

Имя	Значение
NAME	Имя действия. Отображается на кнопке и в названии пункта меню
SMALL_ICON	Место для хранения пиктограммы, которая изображается на кнопке, в пункте меню или на панели инструментов
SHORT_DESCRIPTION	Краткое описание пиктограммы, отображается в строке подсказки
LONG_DESCRIPTION	Подробное описание пиктограммы. Может использоваться для подсказки. Не используется ни одним компонентом из библиотеки Swing
MNEMONIC_KEY	Мнемоническое сокращение. Отображается в пункте меню (см. главу 9)
ACCELERATOR_KEY	Место для хранения сочетания клавиш. Не используется ни одним компонентом из библиотеки Swing
ACTION_COMMAND_KEY	Ранее использовалось в теперь уже устаревшем методе registeredKeyBoardAction()
DEFAULT	Может быть полезным для хранения разнообразных объектов. Не используется ни одним компонентом из библиотеки Swing

Если в меню или к панели инструментов добавляется какое-то действие, его имя и пиктограмма автоматически извлекаются из памяти и отображаются в меню и на панели. Значение SHORT_DESCRIPTION выводится в строке подсказки.

Последние два метода интерфейса Action позволяют извещать другие объекты, в частности меню и панели инструментов, об изменении свойств действий. Например, если меню создано как обработчик, предназначенный для отслеживания изменения свойств действия, и это действие впоследствии было заблокировано, то при отображении меню на экране соответствующее имя действия может быть выделено серым цветом. Обработчики изменения свойств представляют собой часть модели JavaBeans. Подробно компоненты JavaBeans и их свойства будут рассматриваться во втором томе.

Обратите внимание на то, что Action является *интерфейсом*, а не классом. Любой класс, реализующий этот интерфейс, должен реализовать семь методов, которые мы только что рассмотрели. К счастью, это сделать достаточно легко, поскольку все они, кроме первого метода, содержатся в классе AbstractAction. Класс AbstractAction предназначен для хранения пар, состоящих из имен и значений, а также для управления обработчиками изменения свойств. Все, что нужно для этого сделать для практического использования, – это создать подкласс и добавить метод actionPerformed().

Попробуем создать действие, изменяющее цвет фона. Поместим в память имя команды, соответствующую пиктограмму и желаемый цвет. Код цвета запишем в таблицу, состоящую из пар имя-значение, предусмотренных классом AbstractAction. Ниже приводится класс ColorAction. Конструктор задает пары, состоящие из имен и значений, а метод actionPerformed() изменяет цвет фона.

```
public class ColorAction extends AbstractAction
{
   public ColorAction(String name, Icon icon, Color c)
   {
      putValue(Action.NAME, name);
      putValue(Action.SMALL_ICON, icon);
      putValue("color", c);
      putValue(Action.SHORT_DESCRIPTION,
         "Set panel color to " + name.toLowerCase());
   }

   public void actionPerformed(ActionEvent event)
   {
      Color c = (Color) getValue("color");
      setBackground(c);
   }
}
```

В программе создаются три объекта этого класса.

```
Action blueAction = new ColorAction("Blue",
   new ImageIcon("blue-ball.gif"), Color.BLUE);
```

Теперь свяжем это действие с кнопкой. Сделать это можно с помощью конструктора JButton, получающего объект Action в качестве параметра.

```
JButton blueButton = new JButton(blueAction);
```

Этот конструктор считывает имя и пиктограмму действия, помещает короткое описание в строку подсказки и регистрирует объект Action в качестве обработчика. Пиктограмма и строка подсказки показаны на рис. 8.9.

Как мы увидим в следующей главе, точно так же легко можно добавлять действия в меню.

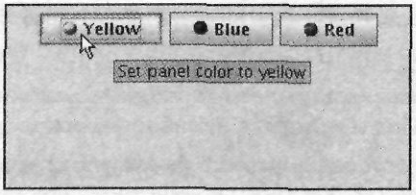


Рис. 8.9. Кнопки с пиктограммами из объекта класса ColorAction, реализующего интерфейс Action

Кроме того, нужно связать действия с нажатием клавиш. Для этого необходимо на время погрузиться в технические детали. Нажатия кнопок передаются компонентам, обладающим фокусом ввода. В нашем примере предусмотрены три компонента – три кнопки на панели. Следовательно, в любой момент одна из этих трех кнопок может владеть фокусом. *Каждая* из этих кнопок должна обрабатывать события, связанные с нажатием клавиш, и реагировать на сочетания клавиш <Ctrl+Y>, <Ctrl+B> и <Ctrl+R>.

Эта проблема имеет универсальный характер, и разработчики пакета Swing нашли ее удачное решение.

На самом деле в версии 1.2 пакета JDK были предложены два решения, позволяющих связать клавиши с действиями: метод `registerKeyboardAction()` класса `JComponent` и механизм отображения клавиш для команд `JTextComponent`. В версии JDK 1.3 решения были объединены. В данном разделе описывается именно этот унифицированный подход.

Для того чтобы связать действия с нажатием клавиш, сначала нужно создать объект класса `KeyStroke`. Это удобный класс, инкапсулирующий описание клавиш. Чтобы создать объект этого класса, не нужно вызывать конструктор. Вместо него следует использовать статический метод `getKeyStroke()` класса `KeyStroke`. Теперь необходимо указать виртуальный код клавиш и флаги (например, для сочетаний, в состав которых входят клавиши <Shift> или <Ctrl>).

```
KeyStroke ctrlBKey = KeyStroke.getKeyStroke(KeyEvent.VK_B,
   InputEvent.CTRL_MASK);
```

Существует также способ, позволяющий описывать нажатие клавиши в виде строки.

```
KeyStroke ctrlBKey = KeyStroke.getKeyStroke("ctrl B");
```

Каждый объект класса `JComponent` содержит три *отображения ввода* (input map), связывающих объекты класса `KeyStroke` с действиями. Эти отображения соответствуют различным условиям (табл. 8.4).

Таблица 8.4. Условия, соответствующие трем отображениям ввода

Флаг	Условие
`WHEN_FOCUSED`	Когда данный компонент находится в фокусе клавиатуры
`WHEN_ANCESTOR_OF_FOCUSED_COMPONENT`	Когда данный компонент содержит другой компонент, находящийся в фокусе клавиатуры
`WHEN_IN_FOCUSED_WINDOW`	Когда данный компонент содержится в том же окне, что и компонент, находящийся в фокусе клавиатуры

При нажатии клавиши отображения ввода обрабатываются в следующем порядке.

1. Проверяется отображение WHEN_FOCUSED компонента, владеющего фокусом ввода. Если предусмотрена реакция на нажатие клавиши, выполняется соответствующее действие, если действие не заблокировано, процесс прекращается.
2. Начиная с компонента, содержащего фокус ввода, выполняется проверка условия WHEN_ANCESTOR_OF_FOCUSED_COMPONENT его родительского компонента. Как только условие будет выполнено, производится соответствующее действие. Если действие не заблокировано, процесс прекращается.
3. Проверяются все видимые и незаблокированные компоненты в окне, обладающем фокусом ввода. Компоненты должны быть зарегистрированы в условии WHEN_IN_FOCUSED_WINDOW. Каждый из этих компонентов (в порядке регистрации) получает возможность выполнить соответствующее действие. Когда будет выполнено первое незаблокированное действие, процесс прекратится. Если клавиша присутствует в нескольких отображениях WHEN_IN_FOCUSED_WINDOW, то результаты, полученные на данном этапе, могут быть неоднозначными.

Отображение ввода извлекается из компонента с помощью метода getInputMap().

```
InputMap imap = panel.getInputMap(JComponent.WHEN_FOCUSED);
```

Условие WHEN_FOCUSED означает, что данное отображение проверяется, если компонент обладает фокусом ввода. В нашем случае это условие не проверяется, поскольку фокусом владеет одна кнопка, а не панель в целом. Каждое из оставшихся двух отображений также позволяет очень легко изменить цвет фона в ответ на нажатие клавиш. В программе мы будем использовать WHEN_ANCESTOR_OF_FOCUSED_COMPONENT.

Класс InputMap не связывает напрямую объекты класса KeyStroke с объектами класса ColorAction, реализующего интерфейс Action. Вместо этого объектам класса KeyStroke он ставит в соответствие произвольные объекты, а с помощью второго отображения, реализованного классом ActionMap, связывает их с действиями. Это облегчает связывание одного и того же действия с разными клавишами, зарегистрированными в разных отображениях ввода.

Итак, каждый компонент имеет три отображения ввода и одно отображение, связывающее нажатие клавиш с действиями. Чтобы объединить их, нужно присвоить каждому действию имя.

```
imap.put(KeyStroke.getKeyStroke("ctrl Y"), "panel.yellow");
ActionMap amap = panel.getActionMap();
amap.put("panel.yellow", yellowAction);
```

Если нужно задать "пустое" действие, можно использовать строку "none". Это позволяет легко заблокировать клавишу.

```
imap.put(KeyStroke.getKeyStroke("ctrl C", "none");
```

В документации JDK предполагается, что название клавиш и соответствующего действия совпадают. Такое решение вряд ли можно назвать оптимальным. Имя действия отображается на кнопке и в пункте меню. Оно может изменяться в процессе разработки, в частности, это неизбежно при адаптации пользовательского интерфейса для разных языков. Мы рекомендуем делать имена действий, которые выводятся на экран, не зависящими от их настоящих имен.

Итак, чтобы одна и та же операция выполнялась в ответ на щелчок на кнопке, выбор пункта меню или нажатие клавиши, надо сделать следующее

1. Создать класс, расширяющий класс `AbstractAction`. Один класс можно будет использовать для программирования разных действий.
2. Создать объект класса, реализующего интерфейс `Action`.
3. Создать кнопку или пункт меню на основе объекта класса, реализующего интерфейс `Action`.
4. Для действий, которые выполняются в ответ на нажатие клавиш, нужно сделать один дополнительный шаг. Сначала следует идентифицировать компонент окна верхнего уровня, например панель, содержащую все остальные компоненты.
5. Проверить отображение `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT` компонента верхнего уровня. Создать объект класса `KeyStroke` для нужного нажатия клавиш. Создать объект, соответствующий нажатию клавиш, например строку, описывающую нужное действие. Добавить пару (нажатие клавиши, действие) к отображению ввода.
6. Получить информацию о соответствии действий и клавиш для компонента верхнего уровня. Добавить пару (клавиша, действие) в отображение.

В листинге 8.5 приведен полный текст программы, задающей соответствие между кнопками, нажатием клавиш и действиями. Действия выполняются в ответ на активизацию кнопок или нажатие клавиш <Ctrl+Y>, <Ctrl+B> или <Ctrl+R>.

Листинг 8.5. Содержимое файла `ActionTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ActionTest
{
   public static void main(String[] args)
   {
      ActionFrame frame = new ActionFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с панелью, демонстрирующий изменение цвета.
*/
class ActionFrame extends JFrame
{
   public ActionFrame()
   {
      setTitle("ActionTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление панели к фрейму.

      ActionPanel panel = new ActionPanel();
```

```
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
   Панель, цвет которой можно изменять щелчком на кнопке или
   нажатием клавиш.
*/
class ActionPanel extends JPanel
{
   public ActionPanel()
   {
      // Определение действий.

      Action yellowAction = new ColorAction("Yellow",
         new ImageIcon("yellow-ball.gif"),
         Color.YELLOW);
      Action blueAction = new ColorAction("Blue",
         new ImageIcon("blue-ball.gif"),
         Color.BLUE);
      Action redAction = new ColorAction("Red",
         new ImageIcon("red-ball.gif"),
         Color.RED);

      // Кнопки, предназначенные для выполнения операций.

      add(new JButton(yellowAction));
      add(new JButton(blueAction));
      add(new JButton(redAction));

      // Связывание клавиш Y, B и R с именами.

      InputMap imap =
       getInputMap(JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);

      imap.put(KeyStroke.getKeyStroke("ctrl Y"), "panel.yellow");
      imap.put(KeyStroke.getKeyStroke("ctrl B"), "panel.blue");
      imap.put(KeyStroke.getKeyStroke("ctrl R"), "panel.red");

      // Связывание имен с действиями

      ActionMap amap = getActionMap();
      amap.put("panel.yellow", yellowAction);
      amap.put("panel.blue", blueAction);
      amap.put("panel.red", redAction);
   }

   public class ColorAction extends AbstractAction
   {
      /**
         Создание действий для изменения цвета.
         @param name Надпись на кнопке
         @param icon Пиктограмма на кнопке
         @param c Цвет фона
```

```
      */
      public ColorAction(String name, Icon icon, Color c)
      {
         putValue(Action.NAME, name);
         putValue(Action.SMALL_ICON, icon);
         putValue(Action.SHORT_DESCRIPTION,
               "Set panel color to " + name.toLowerCase());
         putValue("color", c);
      }

      public void actionPerformed(ActionEvent event)
      {
         Color c = (Color) getValue("color");
         setBackground(c);
      }
   }
}
```

API **javax.swing.Action 1.2**

- `void setEnabled(boolean b)`

  Блокирует и разблокирует действия.

- `boolean isEnabled()`

  Возвращает значение `true`, если действие разблокировано.

- `void putValue(String key, Object value)`

  Помещает пару (имя, значение) в объект действия.

  Параметры `key` Имя, которое должно сохраняться в объекте действия. Оно может быть строкой, но несколько имен имеют специальное значение

  `value` Объект, связанный с именем

- `Object getValue(String key)`

  Возвращает значение из сохраненной пары (имя, значение).

API **javax.swing.JMenu 1.2**

- `JMenuItem add(Action a)`

  Добавляет в меню пункт, вызывающий данное действие. Возвращает добавленный пункт меню.

API **javax.swing.KeyStroke 1.2**

- `static KeyStroke getKeyStroke(char keyChar)`

  Создает экземпляр класса `KeyStroke`, инкапсулирующий нажатие клавиши, которое соответствует событию `KEY_TYPED`.

- `static KeyStroke getKeyStroke(int keyCode, int modifiers)`
- `static KeyStroke getKeyStroke(int keyCode, int modifiers, boolean onRelease)`

Создает объект класса KeyStroke, инкапсулирующий нажатие клавиши, которое соответствует событию KEY_PRESSED или KEY_RELEASED.

Параметры	keyCode	Виртуальный код клавиши
	modifiers	Одно из сочетаний: InputEvent.SHIFT_MASK, InputEvent.CONTROL_MASK, InputEvent.ALT_MASK или InputEvent.META_MASK
	onRelease	Значение true, если код соответствует отпущенной клавише

- `static KeyStroke getKeyStroke(String description)`

  Создает объект класса KeyStroke по описанию, представленному в виде строки. Строка состоит из лексем, разделенных пробелами, в следующем формате.

  1. Лексемы shift control ctrl meta alt button1 button2 button3 преобразовываются в соответствующую битовую маску.
  2. Лексема typed должна сопровождаться символом, например "typed a".
  3. Лексема pressed или released означает, что клавиша нажата или отпущена. (По умолчанию предполагается, что клавиша нажата.)
  4. В противном случае лексема с префиксом VK_ должна соответствовать константе KeyEvent, например INSERT соответствует KeyEvent.VK_INSERT.

  Так, строка "released ctrl Y" соответствует выражению getKeyStroke KeyEvent.VK_Y, Event.CTRL_MASK, true).

API **javax.swing.JComponent 1.2**

- `ActionMap getActionMap()` **1.3**

  Возвращает отображения клавиш в действия.

- `InputMap getInputMap(int flag)` **1.3**

  Получает отображение ввода, которое переводит ключи действий в объекты действий.

Параметр	flag	Одно из значений, представленных в табл. 8.4

Многоадресная передача событий

В предыдущем разделе мы связали несколько источников событий с одним обработчиком. Теперь попробуем сделать противоположное. Все источники событий в библиотеке AWT поддерживают модель *многоадресной передачи событий* (multicast model). Это означает, что одно и то же событие может передаваться нескольким объектам-обработчикам. Многоадресная модель передачи событий полезна, когда событие *потенциально* представляет интерес для нескольких объектов. Нужно лишь добавить несколько обработчиков для одного и того же источника событий, и все зарегистрированные объекты смогут реагировать на эти события.

Рассмотрим простое приложение, в котором используется многоадресная передача событий. Предположим, что, щелкая на кнопке New, в некоем фрейме, пользователь открыл несколько окон. Все окна закрываются кнопкой Close all (рис. 8.10).

Как сказано в документации JDK, "API не гарантирует определенного порядка, в котором события передаются обработчикам, зарегистрированным для данного источника". Поэтому нельзя при создании программ реализовывать логику, зависящую от последовательности передачи событий.

Обработчик события, связанного с кнопкой New, – объект `newListener`, *созданный конструктором класса* `MulticastPanel`, *– формирует новый фрейм в ответ на* щелчок на кнопке.

Однако кнопка Close all класса `MulticastPanel` имеет несколько обработчиков. При каждом вызове метод `makeNewFrame()` добавляет к кнопке Close all еще одного обработчика. Когда пользователь щелкает на кнопке Close all, активизируются все обработчики, и каждый из них закрывает свой фрейм.

Метод `actionPerformed()` удаляет обработчик, связанный ранее с кнопкой Close, поскольку в нем больше нет необходимости (фрейм закрыт).

Полный текст программы приведен в листинге 8.6.

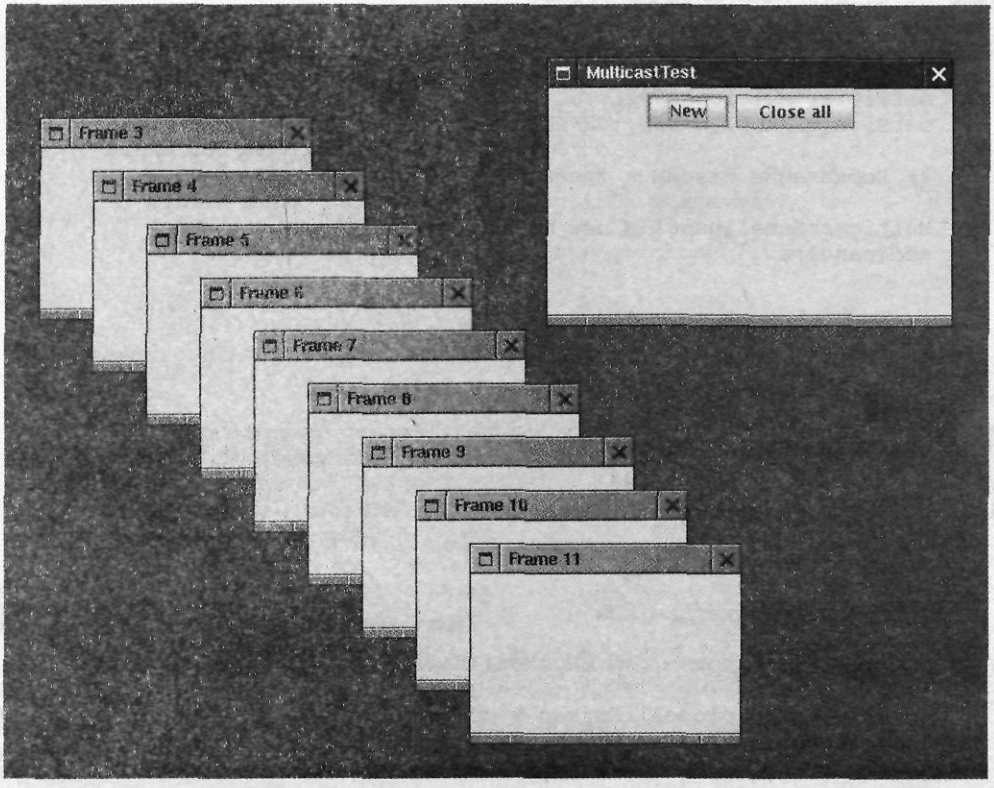


***Рис. 8.10. Все фреймы реагируют на команду* Close all**

Листинг 8.6. Содержимое файла `MulticastTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MulticastTest
{
   public static void main(String[] args)
   {
      MulticastFrame frame = new MulticastFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с кнопками, предназначенными для создания
   и удаления вторичных фреймов.
*/
class MulticastFrame extends JFrame
{
   public MulticastFrame()
   {
      setTitle("MulticastTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление панели к фрейму.

      MulticastPanel panel = new MulticastPanel();
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
   Панель с кнопками для создания и удаления фреймов.
*/
class MulticastPanel extends JPanel
{
   public MulticastPanel()
   {
      // Добавление кнопки New

      JButton newButton = new JButton("New");
      add(newButton);
      final JButton closeAllButton = new JButton("Close all");
      add(closeAllButton);

      ActionListener newListener = new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
```

```
                BlankFrame frame = new BlankFrame(closeAllButton);
                frame.setVisible(true);
            }
         };

      newButton.addActionListener(newListener);
   }
}

/**
   Пустой фрейм, который закрывается щелчком на кнопке.
*/
class BlankFrame extends JFrame
{
   /**
      Создание пустого фрейма.
      @param closeButton Кнопка для закрытия фрейма
   */
   public BlankFrame(final JButton closeButton)
   {
      counter++;
      setTitle("Frame " + counter);
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      setLocation(SPACING * counter, SPACING * counter);

      closeListener = new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               closeButton.removeActionListener(closeListener);
               dispose();
            }
         };
      closeButton.addActionListener(closeListener);
   }

   private ActionListener closeListener;
   public static final int DEFAULT_WIDTH = 200;
   public static final int DEFAULT_HEIGHT = 150;
   public static final int SPACING = 40;
   private static int counter = 0;
}
```

Реализация источников событий

В последнем разделе данной главы мы покажем, как реализуется класс, генерирующий события и оповещающий о них обработчики. Такую задачу приходится решать при использовании сложных компонентов Swing. Как бы то ни было, каждому программисту должен быть интересен механизм связывания обработчиков с компонентами.

В качестве источника событий будет выступать объект `PaintCountPanel`, подсчитывающий число вызовов метода `paintComponent`. При каждом увеличении счетчика `PaintCountPanel` оповещает всех обработчиков. В нашем примере мы свяжем с источником один обработчик, который будет изменять заголовок фрейма (рис. 8.11).

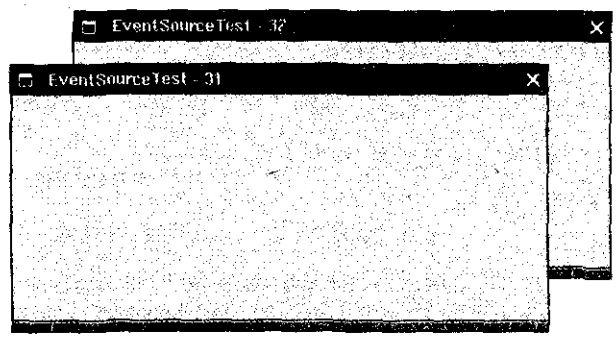


Рис. 8.11. Подсчет числа перерисовок панели

Определяя источник событий, надо указать следующие сведения.

- *Тип события.* Мы можем определить собственный класс события, но для простоты воспользуемся существующим классом `PropertyChangeEvent`.
- *Интерфейс обработчика.* Допустимо определить собственный интерфейс, но мы используем существующий интерфейс `PropertyChangeListener`. В нем объявлен единственный метод.

  ```
  public void propertyChange(PropertyChangeEvent event)
  ```

- *Методы для связывания обработчиков и их удаления.* В классе PaintCountPanel мы определим следующие два метода:

  ```
  public void addPropertyChangeListener
     (PropertyChangeListener listener)
  public void removePropertyChangeListener
     (PropertyChangeListener listener)
  ```

Как убедиться в том, что информация о событиях передана обработчикам? За оповещение отвечает источник. Именно он при возникновении события должен создать соответствующий объект и передать его зарегистрированным обработчикам.

Задачу управления событиями приходится решать достаточно часто, поэтому в библиотеке Swing предусмотрен класс `EventListenerList`. Этот класс упрощает добавление и удаление обработчиков, а также генерацию событий. Он также берет на себя всю сложную работу, связанную с обработкой нескольких потоков, пытающихся одновременно добавлять, удалять или передавать события.

Поскольку некоторые источники событий допускают использование нескольких типов обработчиков, каждый обработчик в списке событий соответствует конкретному классу. Методы `add()` и `remove()` предназначены для реализации методов `addXxxlistener`.

```
public void addPropertyChangeListener
   (PropertyChangeListener listener)
{
   listenerList.add(PropertyChangeListener.class, listener);
}
```

```
public void removePropertyChangeListener
   (PropertyChangeListener listener)
{
   listenerList.remove(PropertyChangeListener.class, listener);
}
```

Может возникнуть вопрос, почему класс `EventListenerList` не проверяет, какой интерфейс реализуется объектом обработчика. Однако объект может реализовывать несколько интерфейсов. Например, может оказаться, что объект, на который ссылается параметр `listener`, одновременно реализует интерфейсы `PropertyChangeListener` и `ActionListener`, а программист захочет, чтобы при вызове метода `addPropertyChangeListener()` добавлялся только обработчик `PropertyChangeListener`. Класс `EventListenerList` должен обеспечить возможность такого выбора.

При вызове метода `paintComponent()` класс `PaintCountPanel` создает объект `PropertyChangeEvent`, указывая в нем источник события, имя свойства, его старое и новое значения. Затем происходит вызов вспомогательного метода `firePropertyChangeEvent()`.

```
public void paintComponent(Graphics g)
{
   int oldPaintCount = paintCount;
   paintCount++;
   firePropertyChangeEvent(new PropertyChangeEvent(this,
      "paintCount", oldPaintCount, paintCount));
   super.paintComponent(g);
}
```

Метод `firePropertyChangeEvent()` обнаруживает всех зарегистрированных обработчиков и для каждого из них вызывает метод `propertyChange()`.

```
public void firePropertyChangeEvent(PropertyChangeEvent event)
{
   EventListener[] listeners =
      listenerList.getListeners(PropertyChangeListener.class);
   for (EventListener l : listeners)
      ((PropertyChangeListener) l).propertyChange(event);
}
```

В листинге 8.7 приведен исходный код программы, основу которой составляет класс `PaintCountPanel`. Конструктор фрейма связывает с панелью обработчик, реагирующий на изменение свойства. Этот обработчик обновляет строку заголовка фрейма.

```
panel.addPropertyChangeListener(new
   PropertyChangeListener()
   {
      public void propertyChange(PropertyChangeEvent event)
      {
         setTitle("EventSourceTest - " + event.getNewValue());
      }
   });
```

Этим примером мы заканчиваем обсуждение обработки событий. В следующей главе подробнее описываются компоненты пользовательского интерфейса. Разумеется, что-

бы запрограммировать пользовательский интерфейс, необходимо знать механизм обработки событий, поскольку действия большинства программ выполняются именно в ответ на события, генерируемые компонентами пользовательского интерфейса.

Листинг 8.7. Содержимое файла `CustomEventTest.java`

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import java.beans.*;

public class EventSourceTest
{
   public static void main(String[] args)
   {
      EventSourceFrame frame = new EventSourceFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм, содержащий панель, которая допускает рисование.
*/
class EventSourceFrame extends JFrame
{
   public EventSourceFrame()
   {
      setTitle("EventSourceTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление панели к фрейму.

      final PaintCountPanel panel = new PaintCountPanel();
      add(panel);

      panel.addPropertyChangeListener(new
         PropertyChangeListener()
         {
            public void propertyChange(PropertyChangeEvent event)
            {
               setTitle("EventSourceTest - " +
                  event.getNewValue());
            }
         });
   }

   public static final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
   Панель, подсчитывающая число перерисовок.
*/
```

```
class PaintCountPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      int oldPaintCount = paintCount;
      paintCount++;
      firePropertyChangeEvent(new PropertyChangeEvent(this,
         "paintCount", oldPaintCount, paintCount));
      super.paintComponent(g);
   }

   /**
      Добавление обработчика событий.
      @param listener Обработчик
   */
   public void addPropertyChangeListener
      (PropertyChangeListener listener)
   {
      listenerList.add(PropertyChangeListener.class, listener);
   }

   /**
      Удаление обработчика событий.
      @param listener Обработчик
   */
   public void removePropertyChangeListener
      (PropertyChangeListener listener)
   {
      listenerList.remove(PropertyChangeListener.class, listener);
   }

   public void firePropertyChangeEvent(PropertyChangeEvent event)
   {
      EventListener[] listeners =
         listenerList.getListeners
         (PropertyChangeListener.class);
      for (EventListener l : listeners)
         ((PropertyChangeListener) l).propertyChange(event);
   }

   public int getPaintCount()
   {
      return paintCount;
   }

   private int paintCount;
}
```

API **javax.swing.event.EventListenerList 1.2**

- `void add(Class t, EventListener l)`

  Добавляет обработчик событий и его класс в список. Класс хранится таким образом, чтобы методы, генерирующие события, могли выбирать требуемый класс. Обычно данный метод используется при построении метода add*Xxx*Listener().

```
public void addXxxListener(XxxListener l)
{
  listenerList.add(XxxListener.class, l);
}
```

Параметры `t` Тип обработчика
`l` Обработчик

- `void remove(Class t, EventListener l)`

  Удаляет обработчик события и его класс из списка. Обычно данный метод используется для создания метода `removeXxxListener()`.

```
public void removeXxxListener(XxxListener l)
{
  listenerList.remove(XxxListener.class, l);
}
```

Параметры `t` Тип обработчика
`l` Обработчик

- `EventListener[] getListeners(Class t)` **1.3**

  Возвращает массив, содержащий всех обработчиков заданного типа. Гарантируется, что метод не вернет значение `null`.

- `Object[] getListenerList()`

  Возвращает массив, в котором элементами с четными индексами являются классы, а с нечетными – объекты-обработчики. Гарантируется, что метод не вернет значение `null`.

java.beans.PropertyChangeEvent 1.1

- `PropertyChangeEvent(Object source, String name, Object oldValue, Object newValue)`

  Создает событие, соответствующее изменению свойства.

Параметры `source` Источник события – объект, сообщающий об изменении свойства
`name` Имя свойства
`oldValue` Значение свойства до изменения
`newValue` Значение свойства после изменения

java.beans.PropertyChangeListener 1.1

- `void propertyChange(PropertyChangeEvent event)`

  Вызывается при изменении значения свойства.

Глава 9

Компоненты пользовательского интерфейса из пакета Swing

В этой главе...

- Архитектура "модель–представление–контроллер"
- Общие сведения о компоновке элементов
- Ввод текста
- Выбор вариантов
- Меню
- Расширенные средства компоновки
- Диалоговые окна

Предыдущая глава была в основном посвящена рассмотрению модели обработки событий Java. Прочитав ее, вы получили сведения, без которых немыслимо создать приложение с графическим интерфейсом. В этой главе мы обсудим самые важные инструменты, позволяющие создавать полнофункциональные пользовательские интерфейсы.

Сначала мы вкратце рассмотрим архитектурные принципы, лежащие в основе пакета Swing. Если вы хотите эффективно использовать современные компоненты пользовательского интерфейса, необходимо разбираться в том, как они функционируют. Затем мы покажем вам, как применять обычные компоненты пользовательского интерфейса из пакета Swing, например поля редактирования, переключатели и меню. Вы узнаете, как использовать возможности диспетчеров компоновки Java для того, *чтобы разместить компоненты в окне, независимо от его конкретных особенностей.* В заключение мы покажем, как с помощью средств Swing создаются диалоговые окна.

В этой главе описываются основные составляющие пакета Swing, например текстовые компоненты, кнопки и полосы прокрутки. Более сложные компоненты будут описаны во втором томе. Тем, кто хочет более детально ознакомиться со средствами Swing, мы можем посоветовать книги Кима Топли (Kim Topley) *Core JFC* и *Core Swing: Advanced Topics* (обе книги выпущены издательством Prentice Hall PTR).

Архитектура "модель–представление–контроллер"

Как мы пообещали, начнем эту главу с раздела, в котором опишем архитектуру компонентов пакета Swing. Прежде чем разъяснить смысл заголовка, выбранного для этого раздела, задумаемся о том, из чего состоят компоненты графического пользовательского интерфейса, например кнопка, флажок опции, поле редактирования или сложное окно управления каталогами. Каждый компонент имеет следующие характеристики.

- *Содержимое*, например состояние кнопки (нажата или нет) или текст в поле редактирования.
- *Внешний вид* (цвет, размер и т.д.).
- *Поведение* (реакция на события).

Даже в таких относительно простых компонентах, как кнопки, эти характеристики тесно связаны между собой. Очевидно, внешний вид кнопки зависит от стиля интерфейса в целом. Кнопка в стиле Metal отличается от кнопки в стиле Windows или Motif. Кроме того, внешний вид зависит от состояния кнопки: нажатая кнопка должна выглядеть иначе, чем обычная. Состояние, в свою очередь, зависит от событий. Если пользователь щелкнул на кнопке, она считается нажатой.

Разумеется, пользователи могут не вдаваться в подробности внутреннего устройства и характеристик кнопки. В конце концов, технические детали – удел программиста, использовавшего эту кнопку в своей программе. Именно он должен хорошо разбираться в ее особенностях. Программисты обязаны разрабатывать элементы пользовательского интерфейса так, чтобы они правильно работали независимо от выбранного стиля.

Для того чтобы помочь в решении подобных задач, разработчики пакета Swing обратились к хорошо известной архитектуре *модель–представление–контроллер* (model-view-controller). В основе этой архитектуры лежат объектные принципы, описанные в главе 5, один из которых гласит: не перегружать объект слишком большим количеством задач. Класс, предусмотренный для кнопки, не обязан делать сразу все. Поэтому стиль компонента связывается с одним объектом, а его содержимое – с другим. Архитектура "модель–представление–контроллер" позволяет достичь этой цели. Следуя ей, нужно реализовать три отдельных класса.

- Класс *модели*, в котором хранится содержимое.
- Класс *представления*, который отображает содержимое.
- Класс *контроллера*, управляющий вводом данных.

Архитектура "модель–представление–контроллер" задает взаимодействие между этими классами. Модель хранит содержимое и не реализует пользовательский интерфейс. Содержимое кнопки тривиально – это небольшой набор флагов, означающих, нажата кнопка или нет, активна или нет и т.д. Содержимое поля редактирования – это строка. Она не совпадает с *представлением* – если содержимое превышает длину поля редактирования, пользователь видит лишь часть отображаемого текста (рис. 9.1).

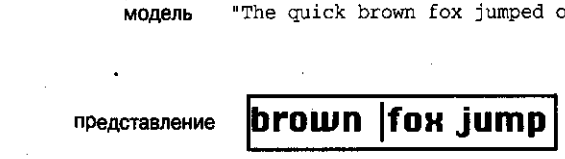


Рис. 9.1. Модель и представление поля редактирования

Модель должна реализовывать методы, изменяющие содержимое и раскрывающие его смысл. Например, модель поля редактирования имеет методы для добавления символов к строке, удаления их и возврата текста в виде строки. Учтите, что модель полностью невидима. Изображать на экране данные, хранящиеся в модели, — задача *представления.*

Термин "модель", видимо, не совсем удачен, поскольку его часто ассоциируют со способом представления абстрактного понятия. Например, авиаконструкторы строят модели для того, чтобы имитировать реальные самолеты. Однако эта аналогия не применима к архитектуре "модель–представление–контроллер". В этом случае модель хранит содержимое, а представление отвечает за ее полное или неполное визуальное изображение. Гораздо более точной аналогией является натурщица, позирующая художнику. Художник должен смотреть на модель и создавать ее изображение. В зависимости от стиля, в котором работает художник, представление может оказаться классическим портретом, картиной в стиле импрессионизма или набором фигур в стиле кубизма.

Одно из преимуществ такой архитектуры состоит в том, что модель может иметь несколько представлений, каждое из которых отражает отдельный аспект ее содержимого. Например, редактор HTML-кода часто предлагает одновременно два представления одних и тех же данных: WYSIWYG (What You See Is What You Get) и набор дескрипторов (рис. 9.2). Когда контроллер обновляет модель, изменяются оба представления. Получив извещение об изменении, представление обновляется автоматически. Разумеется, для таких простых компонентов пользовательского интерфейса, как кнопки, нет никакой необходимости предусматривать несколько представлений одной и той же модели.

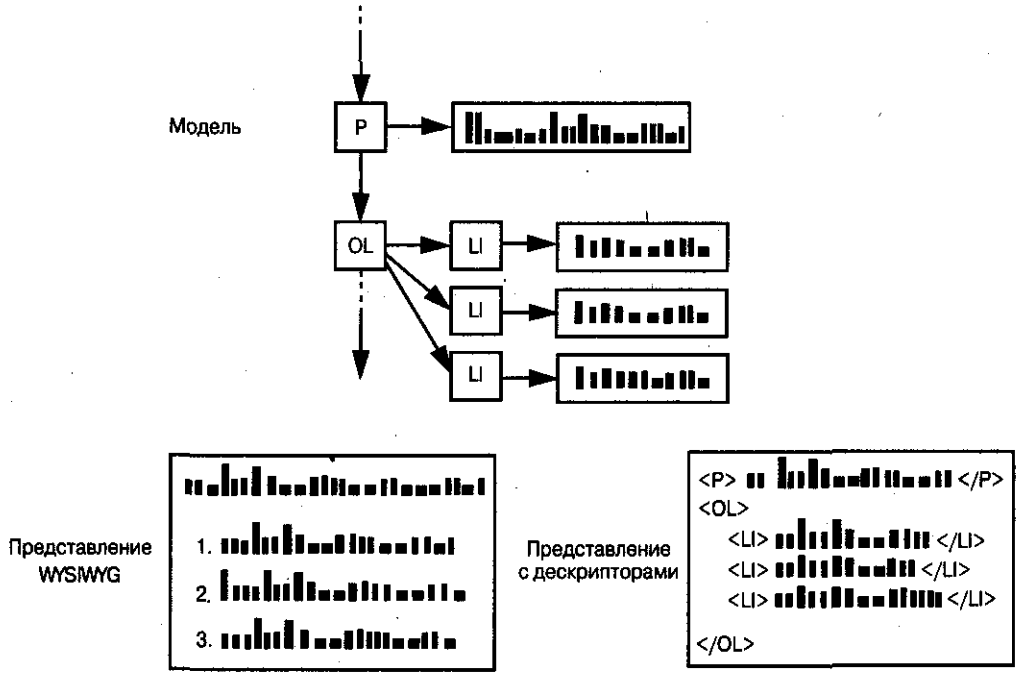


Рис. 9.2. Два разных представления одной и той же модели

Контроллер обрабатывает события, связанные с вводом информации, например щелчки мышью и нажатия клавиш, а затем решает, преобразовывать ли эти события в изменения модели или представления. Например, если пользователь щелкнул на кнопке в текстовом окне, контроллер вызовет команду модели "вставить символ". Затем модель заставит представление обновить изображение. Представление никогда не узнает, почему изменился текст. Однако, если пользователь нажал клавишу, контроллер может отдать представлению команду, вызывающую прокрутку. Прокрутка не изменяет текст, поэтому модель никогда не узнает об этом событии.

Схемы проектирования

Решая задачу, большинство людей руководствуются своим опытом или спрашивают совета у других. Схемы проектирования — это способ представления опыта в структурированном виде.

Последние годы разработчики программного обеспечения стали собирать каталоги таких схем. Пионеров этой области вдохновили схемы проектирования, созданные архитектором Кристофером Александером (Christopher Alexander). Он составил каталог схем для проектирования общественных и частных жилых помещений. Вот типичный пример.

Окна

Каждый любит посидеть в удобном кресле на балконе, у эркера или большого окна с низким подоконником... В комнате, где нет такого места, редко чувствуешь себя удобно и спокойно...

В такой комнате человек хочет удовлетворить два желания:

1. сидеть удобно;
2. сидеть лицом к свету.

Разумеется, если удобные места, т.е. места, где вам больше всего хотелось бы сесть, находятся далеко от окна, то разрешить этот конфликт невозможно.

Следовательно, в каждой комнате, где человек проводит продолжительное время, должно быть по крайней мере одно окно в подходящем для этого месте.

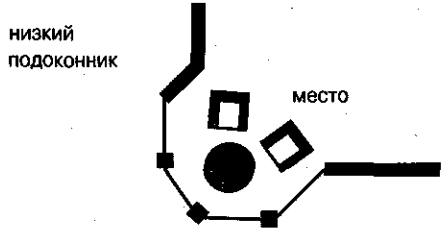


Рис. 9.3. Место у окна

Каждая схема в каталоге Александера, как и в каталоге схем программного обеспечения, следует определенному формату. Сначала описывается контекст, т.е. ситуация, которая порождает проблему. Затем ставится задача, обычно в виде конфликтующих между собой ограничений. В заключение предлагается ее решение, в котором достигается компромисс между противоречивыми требованиями.

В приведенном выше примере контекстом является комната, в которой человек проводит значительную часть дня. Конфликтующие ограничения заключаются в его желании сидеть удобно и при этом лицом к свету. Решение заключается в создании "места у окна".

В архитектуре "модель–представление–контроллер" контекстом является система пользовательского интерфейса, которая отображает информацию и получает данные. В ней существует несколько противоречий. К ним могут относиться разные визуальные представления одних и тех же данных, которые должны одновременно обновляться. Визуальное представление может изменяться в зависимости от стиля интерфейса. Механизмы взаимодействия также могут изменяться, например, для того, чтобы поддерживать выполнение команд, которые вводятся голосом. Решение этой задачи заключается в распределении ответственности между тремя разными взаимодействующими компонентами: моделью, представлением и контроллером.

Разумеется, схема "модель–представление–контроллер" сложнее, чем описанное выше архитектурное решение, касающееся окон, поэтому нужно хорошо изучить механизм взаимодействия ее элементов.

Архитектура "модель–представление–контроллер" — не единственная схема разработки в языке Java. Например, механизм обработки событий из пакета AWT создан на основе схемы наблюдателя.

Схемы разработки стали частью общей культуры программирования. Программисты в любой части света знают, что такое "модель–представление–контроллер" и схема наблюдателя. Таким образом, схемы стали эффективным способом описания проблем разработки программного обеспечения.

Взаимодействие модели, представления и контроллера схематически показано на рис. 9.4.

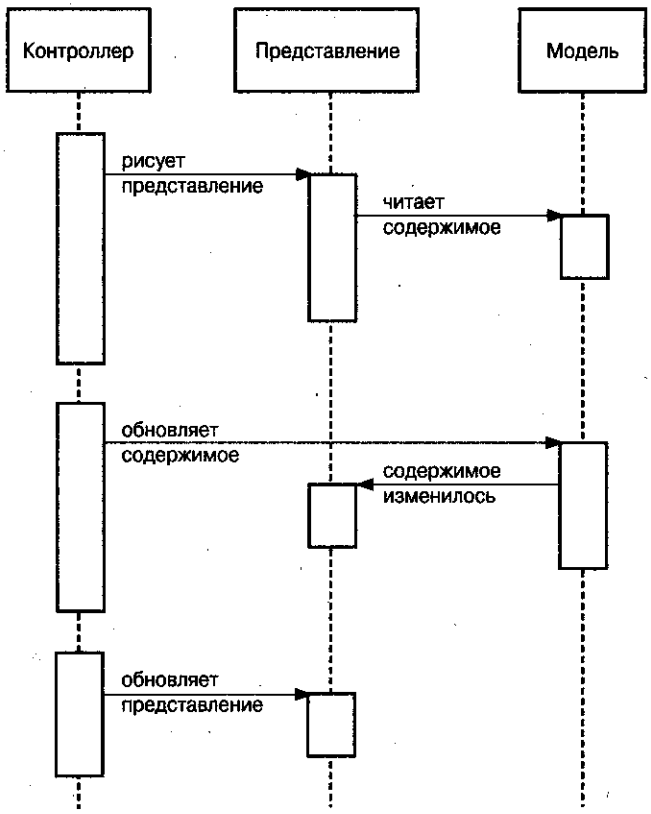


Рис. 9.4. Взаимодействие модели, представления и контроллера

Используя компоненты Swing, совершенно необязательно постоянно думать об архитектуре "модель–представление–контроллер". Каждый пользовательский интерфейс имеет так называемый интерфейсный класс (например, `JButton` или `JTextField`), в котором хранятся модель и ее представление. Если вас интересует содержимое (например, текст в соответствующем поле), интерфейсный класс запросит модель и вернет ответ. Если вас интересует изменение внешнего вида (например, положение курсора в поле редактирования), интерфейсный класс пошлет запрос представлению. Однако иногда интерфейсный класс работает недостаточно хорошо. Тогда нужно извлечь модель и работать непосредственно с ней. (Работать непосредственно с представлением необязательно — это задача кода, задающего стиль.)

Помимо того, что архитектура "модель–представление–контроллер" выполняет возложенные на нее задачи, она оказалась очень удобной для поддержки различных стилей пользовательского интерфейса. Модель кнопки или поля редактирования не зависит от стиля, но от него полностью зависит визуальное представление компонента. Контроллер также может изменяться. Например, в устройствах, управляемых голосом, контроллер должен обрабатывать наборы событий, резко отличающиеся от стандартных событий, связанных с клавиатурой и мышью. Отделяя модель от пользовательского интерфейса, разработчики, использующие пакет Swing, получили возможность повторно применять код модели и даже переключать стиль интерфейса в ходе выполнения программы.

Разумеется, схемы — это лишь руководство к действию, а не догма. Нет такой схемы, которая идеально подходила бы для любых ситуаций. Например, может оказаться сложным следовать схеме "место у окна", если вам нужно перестроить больничную палату. Аналогично, разработчики пакета Swing обнаружили, что в реальной жизни не всегда возможно точно следовать архитектуре "модель–представление–контроллер". Модели легко разделяются, и каждый компонент пользовательского интерфейса имеет свой класс модели. Однако задачи, стоящие перед представлением и контроллером, не всегда четко отделяются одна от другой, поэтому при распределении их между различными классами возникают сложности. Конечно, пользователю этих классов совершенно безразлично, как они реализованы. Как мы уже говорили, о моделях часто можно не думать вовсе, а использовать интерфейсные классы.

Работа кнопок из пакета Swing с точки зрения архитектуры "модель–представление–контроллер"

В предыдущей главе мы показали, как работает кнопка, ничего не говоря о модели, представлении и контроллере. Поскольку кнопка — это простейший элемент пользовательского интерфейса, ее удобно использовать в качестве наглядного пособия для изучения архитектуры "модель–представление–контроллер". При разработке более сложных компонентов используется аналогичный подход.

Для большинства компонентов классы модели реализуют интерфейсы, имена которых оканчиваются словом `Model`. В частности, для кнопок используется интерфейс `ButtonModel`. Классы, реализующие этот интерфейс, могут определять состояние кнопок разного вида. Кнопки настолько просты, что для них в библиотеке Swing предусмотрен отдельный класс `DefaultButtonModel`, реализующий этот интерфейс.

Понять, какого рода данные поддерживаются моделью кнопки, можно, просмотрев методы, объявленные в интерфейсе `ButtonModel` (табл. 9.1).

Таблица 9.1. Методы доступа, объявленные в интерфейсе `ButtonModel`

`getActionCommand()`	Команда, связанная с кнопкой
`getMnemonic()`	Мнемоническое обозначение клавиши или сочетания клавиш, связанного с данной кнопкой
`isArmed()`	Возвращает значение `true`, если кнопка была нажата и курсор мыши все еще находится на кнопке
`isEnabled()`	Возвращает значение `true`, если кнопка не заблокирована
`isPressed()`	Возвращает значение `true`, если кнопка была нажата и кнопка мыши все еще не отпущена
`isRollover()`	Возвращает значение `true`, если курсор мыши находится на кнопке
`isSelected()`	Возвращает значение `true`, если кнопка включена (используется для флажков опций и переключателей)

В каждом объекте `JButton` хранится объект модели кнопки, который можно извлечь оттуда.

```
JButton button = new JButton("Blue");
ButtonModel model = button.getModel();
```

На практике подробности, касающиеся состояния кнопки, интересуют лишь представление, которое рисует ее на экране. Однако из класса `JButton` можно извлечь и другую полезную информацию — в частности, заблокирована ли кнопка. (Для этого класс `JButton` запрашивает ее модель.)

Обратимся еще раз к интерфейсу `ButtonModel` и попробуем определить, *чего в нем не хватает*. Оказывается, модель *не хранит* название кнопки и ее пиктограмму. По этой причине анализ модели не дает информации о внешнем виде кнопки. (При реализации групп переключателей эта особенность может стать источником осложнений.)

Стоит также отметить, что *одна и та же модель* (а именно `DefaultButtonModel`) используется для поддержки кнопок, переключателей, флажков опций и даже для выбора пункта меню. Разумеется, каждый из упомянутых типов кнопок имеет свое собственное представление и отдельный контроллер. Если реализуется интерфейс в стиле Metal, класс `JButton` использует для представления класс `BasicButtonUI`, а для контроллера — класс `ButtonUIListener`. В общем, каждый компонент библиотеки Swing имеет связанный с ним объект представления, название которого заканчивается на `UI`. При этом не все компоненты пакета Swing имеют свой собственный объект контроллера.

Итак, прочитав это краткое вступление, посвященное классу `JButton`, вы можете спросить: а что на самом деле представляет собой класс `JButton`? Это просто интерфейсный класс, подкласс класса `JComponent`, который содержит объект `DefaultButtonModel`, некоторые данные, необходимые для отображения (например, метку кнопки и ее пиктограмму), а также объект класса `DefaultButtonUI`, реализующий представление кнопки.

Общие сведения о компоновке элементов

Прежде чем перейти к обсуждению таких компонентов пакета Swing, как поля редактирования и переключатели, рассмотрим, каким образом они располагаются во фрейме. Поскольку, в отличие от Visual Basic, в JDK нет диспетчера форм, программист должен сам позаботиться о компоновке, т.е. о размещении компонентов пользовательского интерфейса во фрейме.

Разумеется, если вы работаете в интегрированной среде программирования на Java, в ней вполне могут быть предусмотрены средства автоматической компоновки. Несмотря на это, программист обязан четко представлять себе, как осуществляется размещение компонентов, поскольку интерфейсы, полученные с помощью наилучших систем автоматической генерации, обычно нуждаются в ручной "доводке".

Рассмотрим программу из предыдущей главы, в которой кнопки использовались для изменения цвета фрейма (рис. 9.5).

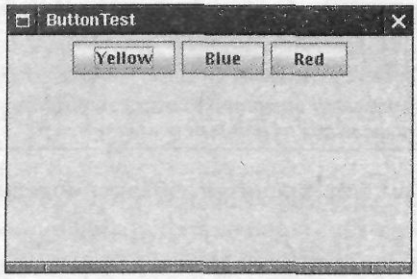


Рис. 9.5. Панель с тремя кнопками

Напомним, как создавалась эта программа.

1. Сначала мы определили внешний вид каждой кнопки, задав строку в конструкторе:

```
JButton yellowButton = new JButton("Yellow");
```

2. Затем включили кнопки в состав панели:

```
add(yellowButton);
```

3. Наконец добавили необходимые обработчики событий:

```
yellowButton.addActionListener(listener);
```

Что произойдет, если к панели добавить еще несколько кнопок? На рис. 9.6 показана панель с шестью кнопками. Расположены они в ряд симметрично относительно центра. Если кнопки не помещаются в строке, они переносятся на новую строку.

Симметрия сохранится, даже если пользователь изменит размер фрейма (рис. 9.7).

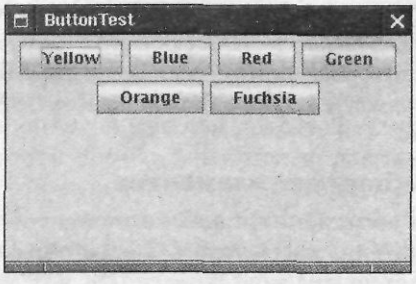


Рис. 9.6. Панель с шестью кнопками, управляемая диспетчером `FlowLAyout`

Рис. 9.7. При изменении размеров панели автоматически изменяется положение кнопок

В языке Java реализована очень элегантная концепция динамической компоновки: все компоненты в составе контейнера управляются *диспетчером компоновки* (layout manager). В нашем примере кнопки управляются диспетчером `FlowLayout`, который связан с панелью по умолчанию.

Диспетчер компоновки `FlowLayout` размещает компоненты горизонтально в ряд, пока хватает места, а затем начинает новую строку.

Если пользователь изменит размеры контейнера, диспетчер компоновки автоматически переместит компоненты, заполняя свободное пространство.

Для размещения компонентов в строке можно выбрать разные способы. По умолчанию интерфейсные элементы размещаются симметрично относительно центра контейнера. Их также можно выровнять по левому или правому краю. Для этого нужно задать параметры `LEFT` или `RIGHT` в конструкторе `FlowLayout`.

```
panel.setLayout(new FlowLayout(FlowLayout.LEFT));
```

Обычно программист предоставляет диспетчеру `FlowLayout` самостоятельно управлять размещением компонентов во фрейме. Однако, используя другой конструктор, можно задать фиксированное расстояние между компонентами (по вертикали и по горизонтали).

`java.awt.Container 1.0`

- `setLayout(LayoutManager m)`

  Задает диспетчер компоновки для данного контейнера.

`java.awt.FlowLayout 1.0`

- `FlowLayout(int align)`

  Создает новый экземпляр класса `FlowLayout` с указанным типом выравнивания.

  Параметры `align` `LEFT`, `CENTER` или `RIGHT`

- `FlowLayout(int align, ing hgap, int vgap)`

  Создает новый объект класса `FlowLayout` с указанными типом выравнивания и расстоянием между компонентами.

  Параметры `align` `LEFT`, `CENTER` или `RIGHT`.

`hgap`	Расстояние между компонентами по горизонтали, заданное в пикселях (при отрицательных значениях компоненты перекрываются)
`vgap`	Расстояние между компонентами по вертикали, заданное в пикселях (при отрицательных значениях компоненты перекрываются)

Диспетчер компоновки BorderLayout

В языке Java есть несколько диспетчеров компоновки. Кроме того, программист может создать свой собственный диспетчер; как это сделать, описывается далее в этой главе. А сейчас рассмотрим диспетчер компоновки `BorderLayout`. Он по умолчанию связывается с панелью содержимого, присутствующей в объекте `JFrame`. В отличие от диспетчера `FlowLayout`, диспетчер `BorderLayout` позволяет выбрать место для каждого компонента. Компонент можно разместить в центре панели, в ее верхней или нижней части, а также слева или справа (рис. 9.8).

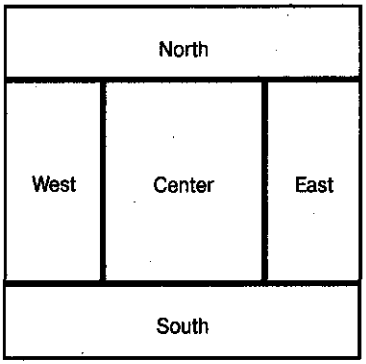


Рис. 9.8. Области, поддерживаемые диспетчером компоновки `BorderLayout`

Ниже приведен пример использования данного диспетчера.

```
panel.setLayout(new BorderLayout());
panel.add(yellowButton, BorderLayout.SOUTH);
```

При размещении компонентов сначала выделяется место по краям контейнера, а оставшееся свободным пространство считается центральной областью. При изменении размера контейнера компонентов, расположенных по краям, остается постоянной; изменяется лишь размер центральной области. При добавлении компонента к панели указываются константы `CENTER`, `NORTH`, `SOUTH`, `EATH` или `WEST`, определенные в классе `BorderLayout`. Занимать все позиции на панели не обязательно. Если не указано никакого значения, по умолчанию принимается значение `CENTER`.

Константы в классе `BorderLayout` определены как строки. Например, константа `BorderLayout.SOUTH` — это строка `"South"`. Многие программисты предпочитают использовать непосредственно строки, поскольку они короче, например `contentPane.add(component, "South")`. Однако, если вы случайно сделаете в строке опечатку, компилятор не распознает этой ошибки.

Диспетчер компоновки BorderLayout масштабирует все компоненты, так чтобы они заполняли все свободное пространство. (Как вы помните, диспетчер FlowLayout оставляет размеры компонента неизменными.) При использовании диспетчера BorderLayout можно задать расстояние между областями, занимаемыми компонентами. Для этого нужно определить соответствующие параметры конструктора класса BorderLayout.

Как уже было сказано, панель содержимого класса JFrame использует диспетчер BorderLayout. До сих пор мы не пользовались этим преимуществом и добавляли компоненты в область, заданную по умолчанию (центр панели). Однако компоненты *можно добавлять и в другие области.*

```
frame.add(yellowButton, BorderLayout.SOUTH);
```

При выполнении данного фрагмента могут возникнуть проблемы, которые мы рассмотрим в следующем разделе.

java.awt.Container 1.0

- `void add(Component c, Object constraints)` **1.1**

  Добавляет компонент к заданному контейнеру.

Параметры	`c`	Добавляемый компонент
	`constraints`	Идентификатор, распознаваемый диспетчером компоновки

java.awt.BorderLayout 1.0

- `BorderLayout(int hgap, int vgap)`

  Создает новый объект класса BorderLayout с фиксированным расстоянием между компонентами.

Параметры	`hgap`	Расстояние по горизонтали между компонентами, заданное в пикселях (при отрицательных значениях компоненты перекрываются)
	`vgap`	Расстояние между компонентами по вертикали, заданное в пикселях (при отрицательных значениях компоненты перекрываются)

Панели

Класс BorderLayout сам по себе не очень интересен. На рис. 9.9 показано, что произойдет, если мы попробуем применить фрагмент кода, приведенный выше. Размеры кнопки увеличатся, и она заполнит собой всю нижнюю часть фрейма. Если мы попытаемся вставить в нижнюю часть еще одну кнопку, она заменит собой предыдущую кнопку.

Один из способов решить эту проблему заключается в использовании дополнительных панелей. Панели представляют собой небольшие контейнеры, содержащие интерфейсные элементы. Эти панели могут, в свою очередь, быть частью более крупной панели, управляемой диспетчером компоновки. Например, одну панель (с кнопками) можно поместить в нижнюю часть, а другую (с текстом) — в центр. Используя

вложенные панели, а также комбинацию диспетчеров `FlowLayout` и `BorderLayout`, можно определять позиции компонентов с достаточной точностью. Этот подход мы будем применять в программах, приведенных в первой части этой главы. Более точный способ размещения компонентов описан ниже в разделе, посвященном классу `GridBagLayout`.

Обратите внимание на рис. 9.10. На панели, расположенной в нижней части панели содержимого, размещены три кнопки.

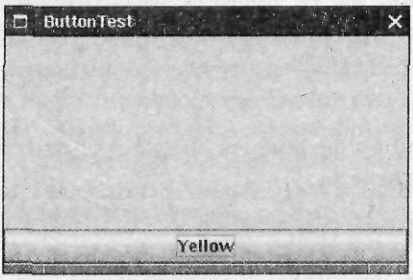


Рис. 9.9. Отдельная кнопка, включенная в контейнер, управляемый диспетчером компоновки `BorderLayout`

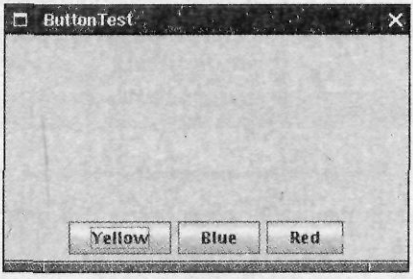


Рис. 9.10. Панель, размещенная в нижней части фрейма

Чтобы получить окно, показанное на рис. 9.9, нам надо добавить к панели содержимого панель с тремя кнопками. Как и следовало ожидать, сначала создается новый экземпляр класса `JPanel`, к которому затем добавляются отдельные кнопки. Как вы уже знаете, по умолчанию с обычной панелью связывается диспетчер компоновки `FlowLayout`. В данной ситуации он вполне подходит. На панель с помощью метода `add()` помещаются отдельные кнопки (эту процедуру мы уже рассматривали ранее). Поскольку мы не переопределяли диспетчер компоновки, то положением кнопок управляет `FlowLayout`. Это значит, что кнопки будут выровнены по центру панели и масштаб их не будет увеличен для заполнения всего свободного пространства. Следующий фрагмент кода добавляет в нижнюю часть контейнера панель, содержащую три кнопки:

```
JPanel panel = new JPanel();
panel.add(yellowButton);
panel.add(blueButton);
panel.add(redButton);
frame.add(panel, BorderLayout.SOUTH);
```

Границы панели невидимы для пользователя. Панели лишь обеспечивают некий механизм, предназначенный для разработки пользовательского интерфейса.

Несмотря на то что класс `JPanel` использует по умолчанию диспетчер `FlowLayout`, в конструкторе класса `JPanel` можно указать другой диспетчер компоновки. Большинство других контейнеров не имеют такого конструктора, однако все они содержат метод `setLayout()`, позволяющий установить диспетчер компоновки, отличающийся от диспетчера, предусмотренного для этого контейнера по умолчанию.

javax.swing.JPanel 1.2

- `JPanel(LayoutManager m)`

  Задает диспетчер компоновки для панели.

Диспетчер компоновки GridLayout

Диспетчер компоновки `GridLayout` размещает компоненты по строкам и столбцам, подобно электронной таблице. Однако при этом размер всех ячеек одинаков. На рис. 9.11 показано окно, в котором для размещения кнопок калькулятора применен диспетчер `GridLayout`. При изменении размера окна кнопки автоматически увеличиваются или уменьшаются, причем размеры всех кнопок остаются одинаковыми.

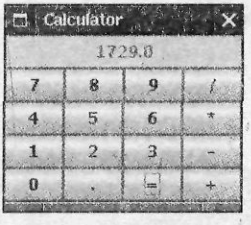


Рис. 9.11. Калькулятор

Требуемое количество строк и столбцов указывается в конструкторе объекта `GridLayout`.

```
panel.setLayout(new GridLayout(5, 4));
```

Как и для других диспетчеров компоновки, при создании объекта `GridLayout` можно указать расстояние между компонентами по горизонтали и по вертикали.

```
panel.setLayout(new GridLayout(5, 4, 3, 3));
```

Два последних параметра в конструкторе задают расстояние между компонентами в пикселях.

Компоненты добавляются построчно, сначала — в первую ячейку первой строки, затем — во вторую ячейку первой строки и т.д.

```
panel.add(new JButton("1"));
panel.add(new JButton("2"));
```

В листинге 9.1 приведен исходный текст программы, имитирующей работу калькулятора. Это обычный калькулятор; он не использует обратную польскую запись, получившую необъяснимую популярность в учебниках по языку Java. В этой программе после включения элементов в состав фрейма мы вызываем метод `pack()`. Он сохраняет рекомендуемый размер компонентов и на их основе вычисляет ширину и высоту фрейма.

На практике лишь немногие программы используют диспетчер компоновки `GridLayout` для размещения компонентов, подобно кнопкам калькулятора. Диспетчер, реализующий сетку из одной строки или из одного столбца, может использоваться для разбиения окна на равные части. Например, если вы хотите разместить в окне ряд кнопок одинакового размера, вам надо включить их в панель, управляемую с помощью диспетчера `GridLayout`. Очевидно, что в конструкторе диспетчера надо задать одну строку, а число столбцов должно равняться количеству кнопок.

Листинг 9.1. Содержимое файла `Calculator.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Calculator
{
   public static void main(String[] args)
   {
      CalculatorFrame frame = new CalculatorFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
    Фрейм с панелью, содержащей кнопки калькулятора.
*/
class CalculatorFrame extends JFrame
{
   public CalculatorFrame()
   {
      setTitle("Calculator");
      CalculatorPanel panel = new CalculatorPanel();
      add(panel);
      pack();
   }
}

/**
   Панель с кнопками калькулятора и элементом
   для отображения результатов вычислений.
*/
class CalculatorPanel extends JPanel
{
   public CalculatorPanel()
   {
      setLayout(new BorderLayout());

      result = 0;
```

```
      lastCommand = "=";
      start = true;

      // Включение элемента для отображения результатов.

      display = new JButton("0");
      display.setEnabled(false);
      add(display, BorderLayout.NORTH);

      ActionListener insert = new InsertAction();
      ActionListener command = new CommandAction();

      // Размещение кнопок в виде сетки 4x4.

      panel = new JPanel();
      panel.setLayout(new GridLayout(4, 4));

      addButton("7", insert);
      addButton("8", insert);
      addButton("9", insert);
      addButton("/", command);

      addButton("4", insert);
      addButton("5", insert);
      addButton("6", insert);
      addButton("*", command);

      addButton("1", insert);
      addButton("2", insert);
      addButton("3", insert);
      addButton("-", command);

      addButton("0", insert);
      addButton(".", insert);
      addButton("=", command);
      addButton("+", command);

      add(panel, BorderLayout.CENTER);
   }

   /**
      Включение кнопки в центральную панель.
      @param label Надпись на кнопке
      @param listener Обработчик активизации кнопки
   */
   private void addButton(String label, ActionListener listener)
   {
      JButton button = new JButton(label);
      button.addActionListener(listener);
      panel.add(button);
   }

   /**
      При обработке события строка, связанная с кнопкой,
      помещается в конец отображаемого текста.
   */
   private class InsertAction implements ActionListener
   {
```

```
   public void actionPerformed(ActionEvent event)
   {
      String input = event.getActionCommand();
      if (start)
      {
         display.setText("");
         start = false;
      }
      display.setText(display.getText() + input);
   }
}

/**
   При обработке события выполняется команда,
   которая определяется строкой, связанной с кнопкой.
*/
private class CommandAction implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      String command = event.getActionCommand();

      if (start)
      {
         if (command.equals("-"))
         {
            display.setText(command);
            start = false;
         }
         else
            lastCommand = command;
      }
      else
      {
         calculate(Double.parseDouble(display.getText()));
         lastCommand = command;
         start = true;
      }
   }
}

/**
   Выполнение вычислений.
   @param x Значение, накапливающее предыдущие результаты
*/
public void calculate(double x)
{
   if (lastCommand.equals("+")) result += x;
   else if (lastCommand.equals("-")) result -= x;
   else if (lastCommand.equals("*")) result *= x;
   else if (lastCommand.equals("/")) result /= x;
   else if (lastCommand.equals("=")) result = x;
   display.setText("" + result);
}

private JButton display;
```

```
   private JPanel panel;
   private double result;
   private String lastCommand;
   private boolean start;
}
```

java.awt.GridLayout 1.0

- `GridLayout(int rows, int cols)`

  Создает новый объект класса `GridLayout`.

Параметры	`rows`	Количество строк в сетке
	`columns`	Количество столбцов в сетке

- `GridLayout(int rows, int cols, int hgap, int vgap)`

  Создает новый объект класса `GridLayout` с заданным расстоянием между объектами по горизонтали и по вертикали.

Параметры	`rows`	Количество строк в сетке
	`columns`	Количество столбцов в сетке
	`hgap`	Расстояние по горизонтали, выраженное в пикселях (отрицательное значение приводит к перекрытию)
	`vgap`	Расстояние по вертикали, выраженное в пикселях (отрицательное значение приводит к перекрытию)

java.awt.Window 1.0

- `void pack()`

  Определяет размеры окна на основании рекомендуемых размеров содержащихся в нем компонентов.

Ввод текста

Теперь мы готовы приступить к изучению интерфейсных элементов, содержащихся в пакете Swing. Начнем с компонентов, позволяющих пользователю вводить и редактировать текст. Для этого в языке Java предусмотрены два компонента: *поле редактирования*, `JTextField`, и *текстовая область*, `JTextArea`. В поле редактирования можно ввести только одну строку, а в текстовую область — несколько.

Классы `JTextField` и `JTextArea` расширяют класс `JTextComponent`. Создать компонент этого класса невозможно, поскольку он является абстрактным. С другой стороны, как это часто бывает при программировании на языке Java, при просмотре документации по интерфейсу прикладных программ оказывается, что существуют методы, которые определены именно в классе `JTextComponent` и лишь наследуются подклассами. В качестве примера можно привести методы, позволяющие поместить текст в поле редактирования или текстовую область, а также извлечь текст из этих компонентов.

javax.swing.text.JTextComponent 1

- `void setText(String t)`

  Изменяет текст в текстовом компоненте.

  Параметр `t` Новый текст

- `String getText()`

  Возвращает текст, содержащийся в данном текстовом компоненте.

- `void setEditable(boolean b)`

  Определяет, может ли пользователь редактировать содержимое компонента.

Поля редактирования

Обычный способ добавить поле редактирования в окно — поместить его на панель или в другой контейнер (точно так же, как и кнопки).

```
JPanel panel = new JPanel();
JTextField textField = new JTextField("Default input", 20);
panel.add(textField);
```

Этот фрагмент кода добавляет поле редактирования, инициализируя его строкой `Default input`. Второй параметр конструктора задает длину строки. В данном случае длина строки равна 20 символам. Если данный параметр равен числу *n*, это означает, что пользователь может ввести в поле редактирования не более *n* символов. К сожалению, символы — не очень точная единица измерения. Их ширина зависит от выбранного шрифта. Ради безопасности длину строки приходится завышать на один или два символа. Кроме того, следует учесть, что заданное количество символов считается пакетом AWT лишь *желательной, или рекомендуемой* длиной строки. Решив уменьшить или увеличить поле редактирования, диспетчер компоновки изменит длину строки. Длина строки, задаваемая параметром конструктора `JTextField`, не является максимально допустимым количеством символов, которое может ввести пользователь в данное поле редактирования. Пользователь может набирать и более длинные строки; если текст выйдет за пределы поля редактирования, произойдет прокрутка содержимого. Часто прокрутка текста плохо воспринимается пользователями, поэтому размер поля редактирования следует задавать с запасом. Если во время выполнения программы возникает необходимость изменить этот размер, следует применить метод `setColumns()`.

Изменив размер поля редактирования с помощью метода `setColumns()`, следует вызвать метод `revalidate()` контейнера, содержащего данный элемент.

```
textField.setColumns(10);
panel.revalidate();
```

Метод `revalidate()` вычисляет заново размер и взаимное расположение всех компонентов в контейнере. Затем диспетчер компоновки перерисовывает контейнер, изменяя размер поля редактирования.

Метод `revalidate()` принадлежит классу `JComponent`. Его выполнение не приводит к немедленному изменению размеров компонента; компонент лишь помечается специальным образом. Такой подход исключает постоянные вычисления при запросе на изменение размеров нескольких компонентов. Если же вы хотите изменить размеры компонентов в составе `JFrame`, вам следует вызвать метод `validate()`; `JFrame` не является подклассом `JComponent`.

Обычно при работе с программой пользователь должен иметь возможность вводить текст или редактировать содержимое поле редактирования. Нередко в начале работы поля редактирования пусты. Для того чтобы создать пустое поле редактирования, не указывайте соответствующий параметр в конструкторе класса `JTextField`.

```
JTextField textField = new JTextField(20);
```

Содержимое поля редактирования можно изменить в любое время, используя метод `setText()` из родительского класса `TextComponent`.

```
textField.setText("Hello!");
```

Как было сказано ранее, определить, какой текст содержится в поле редактирования, можно с помощью метода `getText()`. Чтобы отбросить лишние пробелы до начала текста и после его окончания, надо к значению, возвращаемому методом `getText`, применить метод `trim()`.

```
String text = textField.getText().trim();
```

Для того чтобы задать шрифт для отображения текста, следует использовать метод `setFont()`.

API **javax.swing.JTextField 1.2**

- `JTextField(int cols)`

  Создает пустое поле `JTextField` указанного размера.

  Параметр `cols` Количество символов в поле

- `JTextField(String text, int cols)`

  Создает поле указанного размера, инициализированное заданной строкой.

  Параметры `text` Строка, отображаемая в поле редактирования

  `cols` Количество символов в поле

- `void setColumns(int cols)`

  Задает новый размер поля редактирования, т.е. количество символов, которое должно помещаться в поле.

  Параметр `cols` Количество символов

API **javax.swing.JComponent 1.2**

- `void revalidate()`

  Указывает на то, что размеры компонента и его позиция должны быть вычислены заново.

API **java.awt.Component 1.0**

- `void validate()`

  Вызывает повторное вычисление позиции и размеров компонента. Если компонент является контейнером, позиции содержащихся в нем компонентов также должны быть определены заново.

Метки и маркирующие компоненты

Метки — это компоненты, хранящие текст. Они не имеют обрамления и других видимых элементов. Кроме того, они не реагируют на ввод данных пользователем. Метки можно использовать для идентификации компонентов. Например, в отличие от кнопок, текстовые компоненты не имеют меток, которые позволили бы их различать. Чтобы пометить компонент, не имеющий идентификатора, нужно выполнить следующие действия.

1. Создать компонент `JLabel`, содержащий заданный текст.
2. Поместить его достаточно близко к компоненту, чтобы пользователь мог видеть, что эта метка относится именно к данному интерфейсному элементу.

Конструктор класса `JLabel` позволяет задать текст или пиктограмму, и, при необходимости, выровнять содержимое компонента. Для этой цели используются константы, объявленные в интерфейсе `SwingConstants`. Класс `JLabel` является одним из нескольких классов библиотеки Swing, реализующих этот интерфейс. Таким образом, можно, например, задать метку, текст в которой будет выровнен по левому краю:

```
JLabel label = new JLabel("Minutes", SwingConstants.RIGHT);
```

или

```
JLabel label = new JLabel("Minutes", JLabel.RIGHT);
```

Методы `setText` и `setIcon` позволяют задать текст и пиктограмму метки в ходе выполнения программы.

Начиная с версии J2SE 1.3 на кнопки, метки и пункты меню можно помещать как обычный текст, так и текст в формате HTML. Мы не рекомендуем помещать на кнопки текст в формате HTML — он нарушает стиль интерфейса. Однако в метках такой текст очень эффективен. Строку метки надо поместить между дескрипторами `<HTML>...</HTML>`.

```
label = new JLabel("<html><b>Required</b> entry:</html>");
```

Учтите, что первый компонент с меткой, содержащей текст в формате HTML, отображается на экране с запаздыванием, поскольку для этого нужно загрузить код, выполняющий разбор HTML-выражений.

Метки можно помещать в контейнере. Это означает, что для их размещения применяются такие же подходы, как и для других компонентов.

- `JLabel(String text)`

  Создает метку с текстом, выровненным по левому краю.

  Параметр `text` Текст метки

- `JLabel(Icon icon)`

  Создает метку с пиктограммой, выровненной по левому краю.

  Параметр `icon` Пиктограмма метки

- `JLabel(String text, int align)`

Создает метку, используя заданный текст и тип выравнивания

Параметры `text` Текст метки

`align` Одна из констант: `LEFT`, `CENTER` или `RIGHT`

- `JLabel(String text, Icon icon, int align)`

  Создает метку с текстом и пиктограммой. Пиктограмма располагается левее текста.

  Параметры `text` Текст метки

  `icon` Пиктограмма метки

  `align` Одна из констант: `LEFT`, `CENTER` или `RIGHT`

- `void setText(String text)`

  Задает текст метки.

  Параметр `text` Текст метки

- `void setIcon(Icon icon)`

  Задает пиктограмму метки.

  Параметр `icon` Пиктограмма метки

Отслеживание изменений в полях редактирования

Попробуем создать несколько полей редактирования. На рис. 9.12 показано выполняющееся приложение. Эта программа демонстрирует циферблат часов и два поля редактирования для ввода часов и минут. Как только содержимое полей редактирования изменяется, на циферблате появляется новое время.

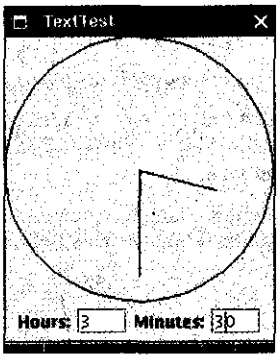


Рис. 9.12. Пример использования полей редактирования

Отслеживать каждое изменение в поле редактирования – довольно трудоемкая задача. *Прежде всего отметим, что контролировать нажатие клавиш в данном случае очень* неудобно. Некоторые из них (например, клавиши со стрелками) не влияют на содержание текста. В зависимости от стиля, на текст могут влиять события, связанные с мышью. Как указывалось в начале этой главы, поле редактирования, предусмотренное в пакете Swing, реализовано в самом общем виде: строка, которую вы видите в поле редактирования, – это просто внешний вид (представление) соответствующей структуры данных (модели). Разумеется, для небольшого поля редактирования различие между его пред-

ставлением и моделью несущественно. Представление – это строка, показанная на экране, а модель – это объект класса `String`. Однако эта же архитектура используется и в компонентах, предназначенных для редактирования форматированного текста, представленного гораздо более сложными структурами данных. Модель для всех текстовых компонентов описывается интерфейсом `Document`, который относится и к обычному, и к форматированному тексту (например, к тексту на языке HTML). Дело в том, что программист может поручить *документу* (не являющемуся текстовым компонентом) сообщить об изменении данных, установив *обработчик документа* (document listener).

```
textField.getDocument().addDocumentListener(listener);
```

При изменении текста вызывается один из трех следующих методов.

```
void insertUpdate(DocumentEvent event)
void removeUpdate(DocumentEvent event)
void changedUpdate(DocumentEvent event)
```

Первые два метода вызываются при вставке или удалении символа. При работе с полями редактирования третий метод не вызывается вовсе. Для более сложных документов ему передается управление в случае выполнения различных модификаций, например изменения формата. К сожалению, не существует способа, который позволил бы сообщить об изменении текста с помощью вызова одного метода. Класс адаптера для этой цели также не предусмотрен. Таким образом, обработчик документа должен реализовывать все три метода. Например, в нашей программе нужен следующий класс:

```
private class ClockFieldListener implements DocumentListener
{
   public void insertUpdate(DocumentEvent event) { setClock(); }
   public void removeUpdate(DocumentEvent event) { setClock(); }
   public void changedUpdate(DocumentEvent event) {}
}
```

Для того чтобы получить строку, введенную пользователем в поле редактирования, метод `setClock()` использует метод `getText()`. К сожалению, этот метод возвращает только строки. После этого их нужно преобразовать в целые числа, применяя довольно громоздкие выражения.

```
int hours = Integer.parseInt(hourField.getText().trim());
int minutes = Integer.parseInt(minuteField.getText().trim());
```

Однако этот фрагмент кода не будет правильно работать, если пользователь введет в поле редактирования последовательность символов, не являющуюся целым числом (например, `two` вместо `2`) или оставит пустое поле. В этом случае метод `parseInt` будет генерировать исключение `NumberFormatException`, и циферблат часов не обновится. В следующем разделе вы узнаете, как можно предотвратить ввод неправильных данных в поле редактирования.

Вместо обработки событий, связанных с документом, обработчик можно непосредственно связать с полем редактирования. Этот обработчик распознает нажатие клавиши <Enter>. Мы не рекомендуем данный подход, поскольку пользователи иногда забывают нажать клавишу <Enter> после ввода данных в поле редактирования. Если вы все же решите поступить подобным образом, вам также надо отслеживать фокус ввода, чтобы обнаружить момент, когда пользователь покинет поле редактирования.

Теперь обратите внимание на то, как конструктор `ClockPanel` устанавливает рекомендуемый размер.

```
public ClockPanel()
{
   setPreferredSize(new Dimension(2 * RADIUS + 1, 2 * RADIUS + 1));
}
```

Теперь, когда метод `pack()` будет вычислять размеры фрейма, он использует при этом рекомендуемые размеры панели.

Листинг 9.2. Содержимое файла `TextTest.java`

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import javax.swing.event.*;

public class TextTest
{
   public static void main(String[] args)
   {
      TextTestFrame frame = new TextTestFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с двумя полями редактирования, предназначенными
   для установки часов.
*/
class TextTestFrame extends JFrame
{
   public TextTestFrame()
   {
      setTitle("TextTest");

      DocumentListener listener = new ClockFieldListener();

      // Добавление панели с полями редактирования.

      JPanel panel = new JPanel();

      panel.add(new JLabel("Hours:"));
      hourField = new JTextField("12", 3);
      panel.add(hourField);
      hourField.getDocument().addDocumentListener(listener);

      panel.add(new JLabel("Minutes:"));
      minuteField = new JTextField("00", 3);
      panel.add(minuteField);
      minuteField.getDocument().addDocumentListener(listener);

      add(panel, BorderLayout.SOUTH);
```

```
      // Добавление циферблата часов.

      clock = new ClockPanel();
      add(clock, BorderLayout.CENTER);
      pack();
   }

   /**
      Установка часов в соответствии с содержимым
      полей редактирования.
   */
   public void setClock()
   {
      try
      {
         int hours = Integer.parseInt(hourField.getText().trim());
         int minutes =
            Integer.parseInt(minuteField.getText().trim());
         clock.setTime(hours, minutes);
      }
      catch (NumberFormatException e) {}
      // Если разбор входных данных не выполнен, часы
      // не устанавливаются
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 300;

   private JTextField hourField;
   private JTextField minuteField;
   private ClockPanel clock;

   private class ClockFieldListener implements DocumentListener
   {
      public void insertUpdate(DocumentEvent event)
         { setClock(); }
      public void removeUpdate(DocumentEvent event)
         { setClock(); }
      public void changedUpdate(DocumentEvent event) {}
   }
}

/**
   Панель для отображения циферблата часов.
*/
class ClockPanel extends JPanel
{
   public ClockPanel()
   {
      setPreferredSize(
         new Dimension(2 * RADIUS + 1, 2 * RADIUS + 1));
   }

   public void paintComponent(Graphics g)
   {
```

```
      // Рисование границы в виде круга.

      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;
      Ellipse2D circle =
         new Ellipse2D.Double(0, 0, 2 * RADIUS, 2 * RADIUS);
      g2.draw(circle);

      // Рисование часовой стрелки.

      double hourAngle =
         Math.toRadians(90 - 360 * minutes / (12 * 60));
      drawHand(g2, hourAngle, HOUR_HAND_LENGTH);

      // Рисование минутной стрелки.

      double minuteAngle =
         Math.toRadians(90 - 360 * minutes / 60);
      drawHand(g2, minuteAngle, MINUTE_HAND_LENGTH);
   }

   public void drawHand(Graphics2D g2,
      double angle, double handLength)
   {
      Point2D end = new Point2D.Double(
         RADIUS + handLength * Math.cos(angle),
         RADIUS - handLength * Math.sin(angle));
      Point2D center = new Point2D.Double(RADIUS, RADIUS);
      g2.draw(new Line2D.Double(center, end));
   }

   /**
      Установка времени.
      @param h Часы
      @param m Минуты
   */
   public void setTime(int h, int m)
   {
      minutes = h * 60 + m;
      repaint();
   }

   private double minutes = 0;
   private int RADIUS = 100;
   private double MINUTE_HAND_LENGTH = 0.8 * RADIUS;
   private double HOUR_HAND_LENGTH = 0.6 * RADIUS;
}
```

API **javax.swing.JComponent 1.2**

- `void setPreferredSize(Dimension d)`

  Устанавливает рекомендуемый размер компонента.

javax.swing.text.Document 1.2

- `int getlength()`

  Возвращает количество символов в документе.
- `String getText(int offset, int length)`

  Возвращает текст, содержащийся в заданной части документа.

  Параметры `offset` Начало текста

  `length` Длина нужной строки
- `void addDocumentListener(DocumentListener listener)`

  Регистрирует обработчика, который оповещается об изменениях в документе.

javax.swing.event.DocumentEvent 1.2

- `Document getDocument()`

  Возвращает документ, являющийся источником события.

javax.swing.event.DocumentListener 1.2

- `void changedUpdate(DocumentEvent e)`

  Вызывается при изменении одного или нескольких атрибутов текста.
- `void insertUpdate(DocumentEvent e)`

  Вызывается при вставке в документ одного или нескольких символов.
- `void removedUpdate(DocumentEvent e)`

  Вызывается при удалении части документа.

Поля для ввода пароля

Поля для ввода пароля представляют собой особый вид полей редактирования. Чтобы скрыть пароль от посторонних наблюдателей, его символы не отображаются на экране. Вместо этого каждый символ в пароле заменяется *эхо-символом* (echo symbol), обычно звездочкой (*). В библиотеке Swing предусмотрен класс `JPasswordField`, реализующий такое поле редактирования.

Поле для ввода пароля – это еще одна демонстрация преимуществ архитектуры "модель-представление-контроллер". Для хранения данных пароль использует ту же модель, что и обычное поле редактирования, но его представление изменено и заменяет все символы пароля эхо-символами.

javax.swing.JPasswordField 1.2

- `JPasswordFirld(String text, int columns)`

  Создает новое поле для ввода пароля.

  Параметры `text` Текст, который должен отображаться на экране. Если текст выводиться не должен, задается значение none

  `columns` Количество символов

- `void setEchoChar(char echo)`

  Задает эхо-символ, который может зависеть от стиля интерфейса. Если задано значение 0, выбирается эхо-символ по умолчанию.

- `char[] getPassword()`

  Возвращает текст, содержащийся в поле для ввода пароля. Для обеспечения большей безопасности возвращаемый массив после использования следует перезаписать. Пароль возвращается как массив символов, а не как объект класса `String`. Причина такого решения заключается в том, что строка может оставаться в виртуальной машине до тех пор, пока не будет уничтожена механизмом сборки мусора.

Поля для форматированного ввода

В предыдущем примере нам необходимо было организовать ввод целых чисел. *Для того чтобы исключить ввод произвольных строк, необходимо ограничить набор* допустимых символов цифрами (0–9) и знаком "минус". Кроме того, знак "минус", если он присутствует, может быть только первым символом строки.

На первый взгляд верификация данных выглядит простой задачей. Можно установить для поля редактирования обработчик нажатий клавиш и затем обрабатывать все события, игнорируя клавиши, не являющиеся цифрами или знаком "минус". К сожалению, этот простой подход на практике себя не оправдывает. Во-первых, не каждое сочетание допустимых символов представляет собой число. Например, последовательности символов `--3` и `3-3` — не числа, хотя и образованы из допустимых символов. Кроме того, существует много способов изменить текст, не нажимая клавиш. В зависимости от особенностей интерфейса, для удаления, копирования и вставки текста могут применяться разные сочетания клавиш. Например, в стиле Metal клавиши <Ctrl+V> используются для вставки содержимого буфера в поле редактирования. Таким образом, приходится следить, чтобы пользователь не вставил из буфера неверные данные. Очевидно, что *попытка отслеживать нажатия клавиш — не самое лучшее решение.*

Как ни странно, до появления пакета JDK 1.4 не существовало компонентов для ввода числовых значений. Начиная с первого издания нашей книги, для реализации поля редактирования, содержащего надлежащим образом форматированное целое число, мы использовали класс `IntTextField`. В каждом новом издании мы изменяли эту реализацию, пытаясь извлечь хоть какую-то пользу из постоянно изменяющихся средств, предназначенных для ограничения вводимых данных. Наконец в JDK 1.4 разработчики реализовали класс `JFormattedTextField`, который можно использовать для ввода не только чисел, но и дат и даже IP-адресов.

Ввод целых чисел

Начнем с простого случая: поля редактирования для ввода целых чисел.

```
JFormattedTextField intField = new
   JFormattedTextField(NumberFormat.getIntegerInstance());
```

Метод `NumberFormat.getIntegerInstance()` возвращает объект для форматирования целых чисел, используя текущие установки для стран и регионов. В США для разделения групп цифр используются запятые. Так, например, пользователь может задавать 1,729 или другие подобные значения. Вопросы интернационализации будут подробно рассмотрены во втором томе.

Как и для любого поля редактирования, мы можем задать длину, т.е. число символов, отображаемых в данном элементе.

```
intField.setColumns(6);
```

Можно также устанавливать значение по умолчанию с помощью метода `setValue()`. Этот метод получает параметр, являющийся объектом класса `Object`, поэтому нам необходимо преобразовать значение типа `int` в объект класса `Integer`.

```
intField.setValue(new Integer(100));
```

Обычно пользователи вводят данные в нескольких полях редактирования, а затем вводят сразу все значения щелчком мыши. Числа, введенные пользователем, считываются с помощью метода `getValue()`. Этот метод возвращает в качестве результата объект класса `Object`, поэтому его необходимо привести к соответствующему типу. Если пользователь изменил значение, метод `JFormattedTextField` вернет объект класса `Long`. Однако, если пользователь не внес никаких изменений, возвращается исходный объект класса `Integer`. Следовательно, возвращаемое значение нужно преобразовать в объект общего суперкласса `Number`.

```
Number value = (Number) intField.getValue();
int v = value.intValue();
```

Поля для форматированного ввода не представляют для нас большого интереса до тех пор, пока пользователь не введет неверные данные. Эту ситуацию мы рассмотрим в следующем разделе.

Поведение поля при потере фокуса ввода

Рассмотрим, что происходит при вводе данных в поле редактирования. Пользователь набирает какой-то текст и вдруг решает прекратить работу с полем, например, щелкнув мышью на другом компоненте. В этом случае поле редактирования *теряет фокус ввода*. Текстовый курсор исчезает из поля, а нажатия клавиш теперь связываются с другим компонентом.

Когда поле редактирования теряет фокус, средства форматирования анализируют текстовую строку, введенную пользователем. Если средства форматирования знают, как преобразовать текстовую строку в объект, текст считается корректным. В противном случае он рассматривается как неправильный. Для проверки корректности текущего содержимого поля редактирования можно применять метод `isEditValid()`.

По умолчанию при потере фокуса ввода поле ведет себя следующим образом. Если текстовая строка является корректной, она *фиксируется*. Средства форматирования превращают ее в объект, который становится текущим значением поля (т.е. значением, возвращаемым методом `getValue()`). Затем это значение преобразовывается обратно в строку, которая становится видимой в поле редактирования. Например, средства форматирования целых чисел признают ввод `123456` корректным, устанавливают текущее значение с помощью оператора `new Long(123456)`, а затем преобразовывают это значение в строку, в которой группы десятичных цифр разделяются запятой: `123,456`.

И наоборот, если текстовая строка некорректна, текущее значение поля не изменяется. В этом случае поле редактирования возвращается в исходное состояние — в нем содержится строка, представляющая старое значение. Например, если пользователь задал в поле редактирования, предназначенном для ввода целых чисел, неверное значение, скажем, `x1`, то будет восстановлено старое значение.

Средства форматирования целых чисел считают текстовую строку корректной, если она начинается с целого числа. Например, строка `1234x` считается корректной. Она преобразуется в число `1234`, а затем представляется как `1,234`.

Поведение поля редактирования можно задать с помощью метода `setFocusLostBehaviour()`. *Фиксация* ("commit") в таком случае несколько отличается от принятой по умолчанию. Если текстовая строка оказывается неверной, то сама строка и значение, записанное в поле, остаются неизменными — не связанными друг с другом. *Устойчивое* ("persist") поведение еще более консервативно. Даже если текстовая строка является правильной, ни сама строка, ни текущее значение поля не изменяется. Для восстановления связи между ними нужно вызывать методы `commitEdit()`, `setValue()` или `setText()`. В заключение опишем *возврат в исходное состояние*, который теперь кажется совершенно излишним. Если поле потеряло фокус, данные, введенные пользователем, игнорируются, а в текстовой строке восстанавливается прежнее значение.

Как правило, описанный выше принцип оправдывает себя. Допустим, что пользователь работает с диалоговым окном и вводит в поле редактирования, предназначенное для ввода целых чисел, строку "1234", поставив перед цифрами пробел, а затем щелкает на кнопке OK. Пробел, стоящий перед цифрами, делает число некорректным, поэтому поле редактирования возвращается в исходное состояние. Обработчик, связанный с кнопкой OK, извлекает значение поля и закрывает диалоговое окно. Пользователь никогда не узнает, что новое значение, введенное им, было отвергнуто. В этой ситуации имеет смысл придерживаться принципа *фиксации* и позволить обработчику перед закрытием диалогового окна проверить, все ли поля содержат корректные значения.

Фильтры

Основные функциональные возможности полей редактирования совершенно очевидны и удовлетворяют требованиям большинства пользователей. Однако их можно усовершенствовать. Допустим, мы хотим предотвратить ввод в поле редактирования символов, не являющихся цифрами. Это поведение реализуется *фильтром документа* (document filter). Напомним, что в архитектуре "модель–представление–контроллер" контроллер преобразует события, связанные с вводом, в команды. Эти команды модифицируют документ, который содержится в поле редактирования, т.е. текстовую строку, хранящуюся в объекте `PlainDocument()`. Например, при обработке команды вставки текста в документ контроллер вызывает команду "вставить строку". В качестве вставляемой строки может выступать либо символ, либо содержимое буфера. Фильтр документа может перехватить эту команду и модифицировать строку либо вовсе отменить вставку. Ниже приведен код фильтра, анализирующего строку, подлежащую вставке, и вставляющего только цифры или символ -.

```
public void insertString(FilterBypass fb, int offset,
            String string, AttributeSet attr)
            throws BadLocationException
   {
      StringBuilder builder = new StringBuilder(string);
      for (int i = builder.length() - 1; i >= 0; i--)
      {
         int cp = builder.codePointAt(i);
```

```
            if (!Character.isDigit(cp) && cp != '-')
            {
               builder.deleteCharAt(i);
               if (Character.isSupplementaryCodePoint(cp))
               {
                  i--;
                  builder.deleteCharAt(i);
               }
            }
         }
         super.insertString(fb, offset, builder.toString(), attr);
      }
```

Следует также переопределить метод `replace()` класса `DocumentFilter` — он вызывается при выделении и замене текста. Реализация метода `replace()` вполне очевидна (см. программу в конце раздела).

Теперь нам необходимо инсталлировать фильтр документа. К сожалению, эта задача не тривиальна. Нужно переопределить метод `getDocumentFilter()` в классе, реализующем средства форматирования, и передать экземпляр этого класса `JformattedTestField`. Поле редактирования, предназначенное для ввода целых чисел, использует `InternationalFormatter`, инициализированный посредством вызова `NumberFormat.getIntegerInstance()`. Ниже показано, как инсталлируются средства форматирования для создания нужного фильтра.

```
JFormattedTextField intField
   = new JFormattedTextField(new
      InternationalFormatter(
         NumberFormat.getIntegerInstance())
      {
         protected DocumentFilter getDocumentFilter()
         {
            return filter;
         }
         private DocumentFilter filter = new IntFilter();
      });
```

В документации JDK указано, что класс `DocumentFilter` был предусмотрен для того, чтобы избежать создания подклассов. В версиях, предшествующих пакету JDK 1.3, фильтрация полей редактирования достигалась путем расширения класса `PlainDocument` и переопределения методов `insertString()` и `replace()`. Теперь класс `PlainDocument` обладает сменными фильтрами. Это очень полезная возможность. Но было бы еще лучше, если бы фильтры были сменными и в классах, реализующих средства форматирования. Увы, этого сделано не было, поэтому нам приходится создавать подклассы этих классов.

Запустите программу `FormatText`, приведенную в конце этого раздела. Третье поле редактирования имеет фильтр. В нем можно вводить только цифры и знак "минус". Обратите внимание на то, что в это поле по-прежнему можно вводить некорректные строки, например `1-2-3`. Как правило, с помощью фильтров нельзя предусмотреть все возможные некорректные строки. Например, строка, состоящая их одного знака "минус" (`-`), некорректна, однако фильтр на это никак не реагирует, поскольку символ "минус" может оказаться префиксом числа `-1`. Хотя фильтры не обеспечивают стопроцентной защиты, их полезно применять для предотвращения явных ошибок.

Фильтры полезно также применять для превращения строчных букв в прописные. Написать такой фильтр довольно легко. Для этого достаточно в методах `insertString()` и `replace()`, принадлежащих фильтру, преобразовать буквы введенной строки, а затем вызвать соответствующий метод суперкласса.

Верификаторы

Для предотвращения неправильного ввода существует еще один полезный механизм. С объектом `JComponent` можно связать *верификатор* (verifier). Если компонент теряет фокус ввода, верификатору посылается запрос. Если в ответ верификатор сообщает, что содержимое компонента некорректно, то фокус немедленно возвращается компоненту. Таким образом, пользователь вынужден исправить содержимое интерфейсного элемента, прежде чем сможет продолжить ввод.

Верификатор должен расширять абстрактный класс `InputVerifier` и определять метод `verify()`. В частности, довольно легко определить верификатор, проверяющий форматированные поля редактирования. Метод `isEditValid()` класса `JFormattedTextField` обращается к средствам форматирования и возвращает значение `true`, если существует способ преобразовать строку в объект. Рассмотрим пример такого верификатора.

```
class FormattedTextFieldVerifier extends InputVerifier
{
   public boolean verify(JComponent component)
   {
      JFormattedTextField field = (JFormattedTextField) component;
      return field.isEditValid();
   }
}
```

Этот верификатор можно связать с любым объектом класса `JFormattedTextField`.

```
intField.setInputVerifier(new FormattedTextFieldVerifier());
```

Однако верификатор еще не гарантирует полной безопасности. Если щелкнуть на кнопке, то это действие будет зарегистрировано обработчиками до того, как некорректный компонент получит фокус. Следовательно, от компонента, не прошедшего верификацию, обработчики могут получить неверный результат. Такое поведение вполне оправдано, поскольку пользователь может щелкнуть на кнопке Cancel, не исправив неверно введенные данные.

Четвертое поле редактирования в программе, приведенной в качестве примера, также связано с верификатором. Попробуйте ввести неверное число (например, `x123`) и нажмите клавишу <Tab> или щелкните мышью на другом поле редактирования. Обратите внимание на то, что фокус будет немедленно возвращен полю. Однако, если щелкнуть на кнопке OK, то обработчик вызовет метод `getValue()`, который сообщит последнее правильное значение.

Прочие стандартные средства форматирования

Кроме средств форматирования целых чисел, класс `JFormatedTextField` поддерживает еще несколько вариантов форматирования. Класс `NumberFormat` содержит статические методы `getNumberInstance()`, `getCurrencyInstance()` и `getPercentInstance()`, обеспечивающие форматирование чисел с плавающей точкой, денежных

знаков и значений, представленных в процентах. Например, можно создать поле редактирования для ввода значений, задающих суммы денег. Для этого нужно сделать следующий вызов:

```
JFormattedTextField currencyField = new
   JFormattedTextField(NumberFormat.getCurrencyInstance());
```

Для того чтобы отредактировать дату и времена, можно воспользоваться методами `getDateInstance()`, `getTimeInstance()` и `getDateTimeInstance()`. Например:

```
JFormattedTextField dateField = new
   JFormattedTextField(DateFormat.getDateInstance());
```

В этом поле можно отредактировать дату в "промежуточном" формате:

```
Feb 24, 2002
```

Можно также выбрать "короткий" формат:

```
2/24/02
```

Для этого надо использовать метод

```
DateFormat.getDateInstance(DateFormat.SHORT)
```

По умолчанию предлагается "снисходительный" формат даты. Иначе говоря, неверная дата, например Февраль 31, 2002, будет преобразована в ближайшую правильную дату — Март 3, 2002. Такая реакция может оказаться неожиданной для пользователей. В этом случае вызывайте метод `setLenient(false)`, принадлежащий объекту `DateFormat`.

С помощью `DefaultFormatter` можно форматировать объекты любого класса, обладающего конструктором со строковым параметром и соответствующим методом `toString()`. Например, класс `URL` имеет конструктор `URL(String)`, который можно использовать для создания URL-адреса из строки.

```
URL url = new URL("http://java.sun.com");
```

Следовательно, `DefaultFormatter` можно применять для форматирования объектов класса `URL`. Для инициализации поля редактирования будет вызван метод `toString()`. Когда поле теряет фокус вода, средства форматирования, используя конструктор с параметром `String`, создают новый объект того класса, которому принадлежит текущее значение. Если конструктор генерирует исключение, редактирование считается некорректным. В этом можно убедиться, попробовав ввести URL, не начинающийся символами "`http:`".

По умолчанию `DefaultFormatter` работает в режиме *замены*. Этим он отличается от остальных средств форматирования. Следует признать, что это не слишком удобно. Для отключения режима замены нужно вызвать метод `setOverwriteMode(false)`.

В заключение отметим, что класс `MaskFormatter` оказывается полезным для создания шаблонов фиксированного размера, состоящих из постоянных и переменных символов. Например, номер карточки социального страхования (скажем, 078-05-1120) можно форматировать следующим образом:

```
new MaskFormatter("###-##-####")
```

Символ # означает отдельную цифру. Символы, которые можно использовать для создания маски, приведены в табл. 9.2.

Таблица 9.2. Символы, используемые для создания маски с помощью `MaskFormatter`

#	Цифра
?	Буква
U	Буква, преобразованная в прописную
L	Буква, преобразованная в строчную
A	Буква или цифра
H	Шестнадцатеричная цифра `[0-9A-Fa-f]`
*	Произвольный символ
'	Управляющий символ

Виды символов для ввода в поле редактирования можно ограничить, вызвав методы `setValidCharacters()` и `setInvalidCharacters()`. Так, для ввода баллов успеваемости (например, A+ или F) нужно использовать следующий вызов:

```
MaskFormatter formatter = new MaskFormatter("U*");
formatter.setValidCharacters("ABCDF+-");
```

Однако способ, позводяющий указать, что второй символ не может быть буквой, отсутствует.

Обратите внимание на то, что строка, форматированная с помощью маски, имеет точно такую же длину, как и сама маска. Если пользователь стирает символы во время редактирования, они заменяются *символами-заполнителями*. По умолчанию символом-заполнителем является пробел, однако можно задать другой символ, вызвав метод `setPlaceholderCharacter()`.

```
formatter.setPlaceholderCharacter('0');
```

По умолчанию средства форматирования с помощью маски работают в режиме вставки, который достаточно удобен, — убедитесь в этом, запустив программу. Отметим также, что курсор "перепрыгивает" через позиции, занятые фиксированными символами маски.

Средства форматирования с помощью маски очень эффективны, если шаблон имеет точно заданную форму, например номер карточки социального страхования или номер телефона. Однако следует заметить, что эти шаблоны не поддаются модификации. Например, невозможно задать маску для международного номера телефона, имеющего переменное количество цифр.

Средства форматирования, определяемые пользователем

Если вам не подходит ни один из стандартных методов форматирования, вы можете легко создать свой собственный. Рассмотрим 4-байтовый IP-адрес:

```
130.65.86.66
```

К нему невозможно применить `MaskFormatter`, поскольку каждый байт может представлять одну, две или три цифры. Кроме того, мы хотели бы проверить значения каждого байта, чтобы они не превышали число 255.

Для того чтобы определить свой собственный метод форматирования, следует расширить класс `DefaktFormatter` и заместить методы `String valueToString(Object value)` и `Object stringToValue(String text)`. Первый метод превращает значение поля в строку, которая отображается в поле редактирования. Второй метод анализирует текст, введенный пользователем, и превращает его в объект. Если один из методов обнаруживает ошибку, генерируется исключение `ParseException`.

В нашей программе IP-адрес хранится в массиве `byte[]`, состоящем из четырех элементов. Метод `valueToString()` формирует строку, в которой байты разделены точками. Обратите внимание на то, что значения элементов массива `byte` имеют знак и лежат в диапазоне от –128 до 127. Чтобы превратить отрицательные байтовые значения в целые числа без знака, следует добавить к ним число 256.

```
public String valueToString(Object value) throws ParseException
{
   if (!(value instanceof byte[]))
      throw new ParseException("Not a byte[]", 0);
   byte[] a = (byte[]) value;
   if (a.length != 4)
      throw new ParseException("Length != 4", 0);
   StringBuilder builder = new StringBuilder();
   for (int i = 0; i < 4; i++)
   {
      int b = a[i];
      if (b < 0) b += 256;
      builder.append(String.valueOf(b));
      if (i < 3) builder.append('.');
   }
   return builder.toString();
}
```

Метод `stringToValue()` анализирует строку и, если строка корректна, создает массив `byte[]`. В противном случае он генерирует исключение `ParseException`.

```
public Object stringToValue(String text) throws ParseException
   {
      StringTokenizer tokenizer = new StringTokenizer(text, ".");
      byte[] a = new byte[4];
      for (int i = 0; i < 4; i++)
      {
         int b = 0;
         try
         {
            b = Integer.parseInt(tokenizer.nextToken());
         }
         catch (NumberFormatException e)
         {
            throw new ParseException("Not an integer", 0);
         }
         if (b < 0 || b >= 256)
            throw new ParseException("Byte out of range", 0);
         a[i] = (byte) b;
      }
      return a;
   }
```

Попробуйте ввести IP-адрес, работая с программой, приведенной в качестве примера. Если вы введете неправильный адрес, в поле будет восстановлено предыдущее корректное значение.

Программа, приведенная в листинге 9.3, демонстрирует различные поля для ввода форматированного текста (рис. 9.13). Чтобы активизировать значения, введенные в полях, щелкните на кнопке OK.

В интерактивной газете *Swing Connection* была опубликована короткая заметка, описывающая метод форматирования, соответствующий любому регулярному выражению (`http://java.sun.com/products/jfc/tsc/articles/reftf`).

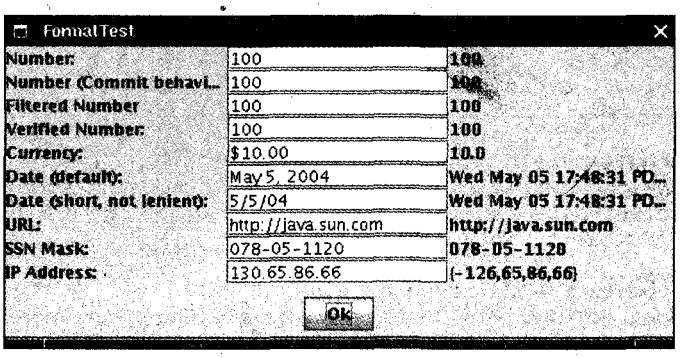


Рис. 9.13. Окно программы `FormatTest`

Листинг 9.3. Содержимое файла `FormatTest.java`

```
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.net.*;
import java.text.*;
import java.util.*;
import javax.swing.*;
import javax.swing.text.*;

/**
   Программа, тестирующая поля для ввода
   форматированного текста.
*/
public class FormatTest
{
   public static void main(String[] args)
   {
      FormatTestFrame frame = new FormatTestFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
```

```
   Фрейм, содержащий поля редактирования с поддержкой формата
   и кнопку, предназначенную для активизации их содержимого.
*/
class FormatTestFrame extends JFrame
{
   public FormatTestFrame()
   {
      setTitle("FormatTest");
      setSize(WIDTH, HEIGHT);

      JPanel buttonPanel = new JPanel();
      okButton = new JButton("OK");
      buttonPanel.add(okButton);
      add(buttonPanel, BorderLayout.SOUTH);

      mainPanel = new JPanel();
      mainPanel.setLayout(new GridLayout(0, 3));
      add(mainPanel, BorderLayout.CENTER);

      JFormattedTextField intField = new
         JFormattedTextField(NumberFormat.getIntegerInstance());
      intField.setValue(new Integer(100));
      addRow("Number:", intField);

      JFormattedTextField intField2 = new
         JFormattedTextField(NumberFormat.getIntegerInstance());
      intField2.setValue(new Integer(100));
      intField2.setFocusLostBehavior(JFormattedTextField.COMMIT);
      addRow("Number (Commit behavior):", intField2);

      JFormattedTextField intField3 = new
        JFormattedTextField(new
        InternationalFormatter(NumberFormat.getIntegerInstance())
            {
               protected DocumentFilter getDocumentFilter()
               {
                  return filter;
               }
               private DocumentFilter filter = new IntFilter();
            });
      intField3.setValue(new Integer(100));
      addRow("Filtered Number", intField3);

      JFormattedTextField intField4 = new
         JFormattedTextField(NumberFormat.getIntegerInstance());
      intField4.setValue(new Integer(100));
      intField4.setInputVerifier(new
         FormattedTextFieldVerifier());
      addRow("Verified Number:", intField4);

      JFormattedTextField currencyField = new
         JFormattedTextField(NumberFormat.getCurrencyInstance());
      currencyField.setValue(new Double(10));
      addRow("Currency:", currencyField);

      JFormattedTextField dateField = new
```

```
      JFormattedTextField(DateFormat.getDateInstance());
   dateField.setValue(new Date());
   addRow("Date (default):", dateField);

   DateFormat format =
      DateFormat.getDateInstance(DateFormat.SHORT);
   format.setLenient(false);
   JFormattedTextField dateField2 = new
      JFormattedTextField(format);
   dateField2.setValue(new Date());
   addRow("Date (short, not lenient):", dateField2);

   try
   {
      DefaultFormatter formatter = new DefaultFormatter();
      formatter.setOverwriteMode(false);
      JFormattedTextField urlField = new
         JFormattedTextField(formatter);
      urlField.setValue(new URL("http://java.sun.com"));
      addRow("URL:", urlField);
   }
   catch (MalformedURLException e)
   {
      e.printStackTrace();
   }

   try
   {
      MaskFormatter formatter = new
         MaskFormatter("###-##-####");
      formatter.setPlaceholderCharacter('0');
      JFormattedTextField ssnField = new
         JFormattedTextField(formatter);
      ssnField.setValue("078-05-1120");
      addRow("SSN Mask:", ssnField);
   }
   catch (ParseException exception)
   {
      exception.printStackTrace();
   }

   JFormattedTextField ipField = new
      JFormattedTextField(new IPAddressFormatter());
   ipField.setValue(new byte[] { (byte) 130, 65, 86, 66 });
   addRow("IP Address:", ipField);
}

/**
   Помещает строку на главную панель.
   @param labelText Метка
   @param field Поле, используемое в качестве примера
*/
public void addRow(String labelText,
   final JFormattedTextField field)
{
   mainPanel.add(new JLabel(labelText));
```

```
      mainPanel.add(field);
      final JLabel valueLabel = new JLabel();
      mainPanel.add(valueLabel);
      okButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               Object value = field.getValue();
               if (value.getClass().isArray())
               {
                  StringBuilder builder = new StringBuilder();
                  builder.append('{');
                  for (int i = 0; i < Array.getLength(value); i++)
                  {
                     if (i > 0) builder.append(',');
                     builder.append(
                        Array.get(value, i).toString());
                  }
                  builder.append('}');
                  valueLabel.setText(builder.toString());
               }
               else
                  valueLabel.setText(value.toString());
            }
         });
   }

   public static final int WIDTH = 500;
   public static final int HEIGHT = 250;

   private JButton okButton;
   private JPanel mainPanel;
}

/**
   Фильтр, ограничивающий ввод цифрами и знаком "минус".
*/
class IntFilter extends DocumentFilter
{
   public void insertString(FilterBypass fb,
      int offset, String string, AttributeSet attr)
      throws BadLocationException
   {
      StringBuilder builder = new StringBuilder(string);
      for (int i = builder.length() - 1; i >= 0; i--)
      {
         int cp = builder.codePointAt(i);
         if (!Character.isDigit(cp) && cp != '-')
         {
            builder.deleteCharAt(i);
            if (Character.isSupplementaryCodePoint(cp))
            {
               i--;
               builder.deleteCharAt(i);
            }
```

```
            }
         }
         super.insertString(fb, offset, builder.toString(), attr);
      }

      public void replace(FilterBypass fb,
         int offset, int length, String string,
         AttributeSet attr)
         throws BadLocationException
      {
         if (string != null)
         {
            StringBuilder builder = new StringBuilder(string);
            for (int i = builder.length() - 1; i >= 0; i--)
            {
               int cp = builder.codePointAt(i);
               if (!Character.isDigit(cp) && cp != '-')
               {
                  builder.deleteCharAt(i);
                  if (Character.isSupplementaryCodePoint(cp))
                  {
                     i--;
                     builder.deleteCharAt(i);
                  }
               }
            }
            string = builder.toString();
         }
         super.replace(fb, offset, length, string, attr);
      }
   }

   /**
      Верификатор, проверяющий корректность содержимого
      полей редактирования.
   */
   class FormattedTextFieldVerifier extends InputVerifier
   {
      public boolean verify(JComponent component)
      {
         JFormattedTextField field = (JFormattedTextField) component;
         return field.isEditValid();
      }
   }

   /**
      Средство форматирования 4-байтовых IP-адресов в виде a.b.c.d.
   */
   class IPAddressFormatter extends DefaultFormatter
   {
      public String valueToString(Object value)
         throws ParseException
      {
         if (!(value instanceof byte[]))
            throw new ParseException("Not a byte[]", 0);
         byte[] a = (byte[]) value;
```

Обычно при работе с программой пользователь должен иметь возможность вводить текст или редактировать содержимое поле редактирования. Нередко в начале работы поля редактирования пусты. Для того чтобы создать пустое поле редактирования, не указывайте соответствующий параметр в конструкторе класса `JTextField`.

```
JTextField textField = new JTextField(20);
```

Содержимое поля редактирования можно изменить в любое время, используя метод `setText()` из родительского класса `TextComponent`.

```
textField.setText("Hello!");
```

Как было сказано ранее, определить, какой текст содержится в поле редактирования, можно с помощью метода `getText()`. Чтобы отбросить лишние пробелы до начала текста и после его окончания, надо к значению, возвращаемому методом `getText`, применить метод `trim()`.

```
String text = textField.getText().trim();
```

Для того чтобы задать шрифт для отображения текста, следует использовать метод `setFont()`.

javax.swing.JTextField 1.2

- `JTextField(int cols)`

  Создает пустое поле `JTextField` указанного размера.

  Параметр `cols` Количество символов в поле

- `JTextField(String text, int cols)`

  Создает поле указанного размера, инициализированное заданной строкой.

  Параметры `text` Строка, отображаемая в поле редактирования

  `cols` Количество символов в поле

- `void setColumns(int cols)`

  Задает новый размер поля редактирования, т.е. количество символов, которое должно помещаться в поле.

  Параметр `cols` Количество символов

javax.swing.JComponent 1.2

- `void revalidate()`

  Указывает на то, что размеры компонента и его позиция должны быть вычислены заново.

java.awt.Component 1.0

- `void validate()`

  Вызывает повторное вычисление позиции и размеров компонента. Если компонент является контейнером, позиции содержащихся в нем компонентов также должны быть определены заново.

- `void setValue(Object value)`

  Пытается присвоить значение указанному объекту. Попытка оказывается безуспешной, если невозможно преобразовать объект в строку.

- `void commitEdit()`

  Пытается присвоить полю корректное значение, вычисленное на основе редактируемой строки. Если преобразовать объект в строку невозможно, попытка окажется безуспешной.

- `boolean isEditValid()`

  Проверяет, корректное ли значение представляет редактируемая строка.

- `void setFicusLostBehaviour(int behaviour)`
- `int getFocusLostBehaviour()`

  Задает поведение поля при потере фокуса ввода или возвращает информацию о поведении. Поведение задается константами `COMMIT_OR_REVERT`, `REVERT`, `COMMIT` и `PERSIST`, определенными в классе `JFormattedTextField`.

javax.text.DateFormat 1.1

- `static DateFormat getDateInstance()`
- `static DateFormat getDateInstance(int dateStyle)`
- `static DateFormat getTimeInstance()`
- `static DateFormat getTimeInstance(int timeStyle)`
- `static DateFormat getDateTimeInstance()`
- `static DateFormat getDateTimeInstance(int dateStyle, int timeStyle)`

Эти методы возвращают объекты класса `DateFormat`, контролирующие дату, время или дату и время одновременно. Значения параметров `dateStyle` и `timeStyle` задаются константами `SHORT`, `MEDIUM`, `LONG`, `FULL` и `DEFAULT`, определенными в классе `DateFormat`.

javax.swing.JFormattedTextField.AbstractFormatter 1.4

- `abstract String valueToString(Object value)`

  Преобразовывает значение в редактируемую строку. Если для параметра `value` не существует соответствующего способа форматирования, генерируется исключение `ParseException`.

- `abstract Object stringToValue(String s)`

  Преобразовывает строку в значение. Если строка `s` не соответствует формату, генерируется исключение `ParseException`.

- `DocumentFilter getDocumentFilter()`

  Этот метод переопределяется для создания фильтра документа, ограничивающего ввод в поле редактирования. Если фильтр не нужен, возвращает значение `null`.

javax.swing.text.DefaultFormatter 1.3

- `void setOverwriteMode(boolean mode)`
- `boolean getOverwriteMode()`

Устанавливает режим замены или возвращает информацию о режиме. Если параметр `mode` имеет значение `true`, то новые символы замещают символы редактируемой строки.

javax.swing.text.DocumentFilter 1.4

- `void insertString(DocumentFilter.FilterByPass bypass, int offset, String text, AttributeSet attrib)`

Вызывается перед включением строки в документ. В случае необходимости можно переопределить этот метод так, чтобы он модифицировал строку. Чтобы запретить включение строки, надо отказаться от вызова `super.insertString()`. При вызове методов `bypass` документ будет модифицирован без фильтрации.

Параметры	`bypass`	Объект, позволяющий выполнять команды редактирования, минуя фильтр
	`offset`	Смещение, с которого начинается вставка текста
	`text`	Вставляемые символы
	`attrib`	Атрибуты форматирования вставляемого текста

- `void replace(DocumentFilter.FilterBypass bypass, int offset, int length, String text, AttributeSet attrib)`

Этот метод вызывается перед заменой части документа новой строкой. По мере необходимости можно переопределить этот метод так, чтобы он модифицировал строку. Чтобы запретить замену, надо отказаться от вызова `super.replace()`. При вызове методов `bypass` документ будет модифицирован без фильтрации.

Параметры	`bypass`	Объект, позволяющий выполнять команды редактирования, минуя фильтр
	`offset`	Смещение, с которого начинается вставка текста
	`length`	Длина заменяемой части текста
	`text`	Вставляемые символы
	`attrib`	Атрибуты форматирования вставляемого текста

- `void remove(DocumentFilter.FilterBypass bypass, int offset, int length)`

Этот метод вызывается перед удалением части документа. Для проверки результата можно вызвать метод `bypass.getDocument()`.

Параметры	`bypass`	Объект, позволяющий выполнять команды редактирования, минуя фильтр
	`offset`	Смещение, с которого начинается удаление текста
	`length`	Длина удаляемой части текста

API javax.swing.text.MaskFormatter 1.4

- `MaskFormatter(String mask)`

  Создает средство форматирования, использующее заданную маску. Символы маски см. в табл. 9.2.

- `void setValidCharacter(String charactyers)`
- `String getValidCharacters()`

  Задает или возвращает допустимые символы. В качестве переменной части маски могут использоваться только символы заданной строки.

- `void setInvalidCharacters(String characters)`
- `String getInvalidCharacters()`

  Задает и возвращает некорректные символы. Ни один из символов заданной строки при вводе не принимается.

- `void setPlaceholderCharacter(char ch)`
- `char getPlaceholderCharacter()`

  Задает и возвращает символ-заполнитель, отображаемый в позиции, в которой пользователь еще не ввел данные. По умолчанию таким символом является пробел.

- `void setPlaceholder(String s)`
- `String getPlaceholder()`

  Задает и возвращает строку, которая используется для заполнения той части маски, которую пользователь еще не ввел. Если вместо строки указано значение `null` или строка короче маски, то для заполнения оставшейся части используется символ-заполнитель.

- `void setValueContainsLiteralCharacters(boolean b)`
- `boolean getValueContainsLiteralCharacters()`

  Задает и возвращает флаг, определяющий использование литеральных символов. Если логическое значение, указываемое в качестве параметра, равно `true`, то значение поля содержит литеральную (постоянную) часть маски. В противном случае литеральные символы удаляются. По умолчанию принимается значение параметра `true`.

Текстовые области

Иногда при работе программы возникает необходимость ввести несколько строк. Как указывалось ранее, для этого применяется компонент `JTextArea`. Поместив данный компонент в свою программу, разработчик предоставляет пользователю возможность вводить любое количество текста, разделяя строки нажатием клавиши <Enter>. Каждая строка заканчивается символом '\n', как это предусмотрено в языке Java. Если возникает необходимость разбить текст, введенный пользователем, на несколько частей и поместить каждую из них в отдельный объект `String`, можно использовать класс `StringTokenizer` (см. главу 12). Пример текстовой области показан на рис. 9.14.

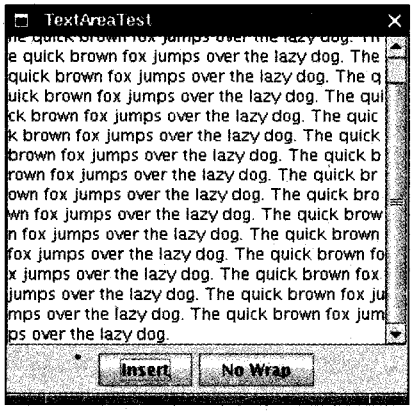


Рис. 9.14. Текстовая область

В конструкторе компонента `JTextArea` указывается количество строк и их длина.

```
textArea = new JTextArea(8, 40); // 8 строк по 40 символов.
```

Параметр, задающий количество символов в строке, действует так же, как и для поля редактирования; его значение рекомендуется немного завысить. Пользователь не ограничен количеством строк и их длиной. Если длина строки или число строк выйдет за пределы заданных параметров, текст будет прокручиваться в окне. Для изменения длины строк можно применять метод `setColumns()`, а для изменения их количества — метод `setRows()`. Эти параметры задают лишь рекомендуемые размеры — диспетчер компоновки может самостоятельно увеличивать или уменьшать размеры поля редактирования.

Если пользователь ввел больше текста, чем помещается в текстовой области, то произойдет отсечение. Этого можно избежать, установив автоматический перенос строки (line-wrapping).

```
textArea.setLineWrap(true); // В длинных строках
                            // выполняется перенос.
```

Автоматический перенос проявляется лишь визуально. Текст, хранящийся в документе, изменен не будет — в него не вставляются символы `'\n'`.

В пакете Swing текстовая область не имеет полос прокрутки. Если они необходимы, текстовую область следует включить в *панель прокрутки* (scroll pane).

```
textArea = new JtextArea(8, 40);
JScrollPane scrollPane = new JScrollPane(textArea);
```

Теперь панель прокрутки управляет представлением текстовой области. Полоса прокрутки автоматически появляется, когда текст выходит за рамки отведенной для него области, и исчезает, когда "лишняя" часть текста удаляется. Прокрутка поддерживается панелью прокрутки — сама программа не обязана обрабатывать события, связанные с ней.

Чтобы снабдить компонент полосами прокрутки, поместите его на панель прокрутки. Это решение универсально и применяется при работе с пакетом Swing очень часто.

Листинг 9.4 содержит полный текст программы, демонстрирующей работу с текстовой областью. Эта программа позволяет пользователю редактировать содержимое текстовой области. Чтобы вставить предложение в конец текста, щелкните на кнопке Insert. Для того чтобы установить или отключить автоматический перенос, щелкните на второй кнопке. (Ее название изменяется с Wrap на No wrap, и наоборот.) Разумеется, для редактирования текста можно использовать только клавиатуру, не прибегая к помощи мыши. Обратите внимание на то, как можно выделить часть текста, вырезать его, скопировать и вставить с помощью сочетаний клавиш <Ctrl+X>, <Ctrl+C> и <Ctrl+V>. (Сочетания клавиш должны соответствовать выбранному стилю интерфейса. В частности, указанные комбинации характерны для стилей Metal, Windows и Mac.)

Компонент `JTextArea` отображает только простой текст, не содержащий специальных шрифтов или особого форматирования. Чтобы отобразить на экране форматированный текст (например, в формате HTML или RTF), можно использовать классы `JEditPane` и `JTextPane`. Эти классы будут обсуждаться во втором томе.

Листинг 9.4. Содержимое файла `TextAreaTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TextAreaTest
{
   public static void main(String[] args)
   {
      TextAreaFrame frame = new TextAreaFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм, содержащий текстовую область и кнопки для редактирования
*/
class TextAreaFrame extends JFrame
{
   public TextAreaFrame()
   {
      setTitle("TextAreaTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      buttonPanel = new JPanel();

      // Добавление кнопки, предназначенной для включения
      // текста в текстовую область.

      JButton insertButton = new JButton("Insert");
      buttonPanel.add(insertButton);
      insertButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
```

```
            textArea.append(
               "The quick brown fox jumps over the lazy dog.");
         }
      });

      // Добавление кнопки, предназначенной для включения
      // и отключения режима переноса.

      wrapButton = new JButton("Wrap");
      buttonPanel.add(wrapButton);
      wrapButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               boolean wrap = !textArea.getLineWrap();
               textArea.setLineWrap(wrap);
               scrollPane.revalidate();
               wrapButton.setText(wrap ? "No Wrap" : "Wrap");
            }
         });

      add(buttonPanel, BorderLayout.SOUTH);

      // Связывание текстовой области с полосами прокрутки.

      textArea = new JTextArea(8, 40);
      scrollPane = new JScrollPane(textArea);

      add(scrollPane, BorderLayout.CENTER);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 300;

   private JTextArea textArea;
   private JScrollPane scrollPane;
   private JPanel buttonPanel;
   private JButton wrapButton;
}
```

API **javax.swing.JTextArea 1.2**

- `JTextArea(int rows, int cols)`

  Создает новую текстовую область.

  Параметры `rows` Количество строк
  `cols` Количество символов в строке

- `JTextArea(String text, int rows, int cols)`

  Создает новую текстовую область, содержащую заданный текст.

  Параметры `text` Заданный текст
  `rows` Количество строк
  `cols` Количество символов в строке

- `void setColumns(int cols)`

  Задает длину строки.

  Параметр `cols` *Количество символов в строке*

- `void setRows()`

  Задает количество строк.

  Параметр `rows` Количество строк

- `void append(String newText)`

  Добавляет заданный текст в конец содержимого текстовой области.

  Параметр `text` Текст, который нужно добавить

- `void setLineWrap(boolean wrap)`

  Включает и отключает режим автоматического переноса строк.

  Параметр `wrap` Значение `true`, если должен выполняться автоматический перенос строк

- `void setWrapStyleForward(boolean word)`

  Если параметр `word` равен `true`, то перенос в длинных строках выполняется по границе слова, в противном случае границы слов во внимание не принимаются.

- `void setTabSize(int c)`

  Устанавливает табуляторы через каждые c символов. Обратите внимание на то, что пробелы не включаются; текст лишь выравнивается по следующей позиции табуляции.

  Параметр `c` Шаг табуляции

API **`javax.swing.JScrollPane 1.2`**

- `JScrollPane(Component c)`

  Создает панель прокрутки, которая отображает содержание указанного компонента. Полоса прокрутки появляется, только если текст превышает допустимые размеры.

  Параметры `c` Компонент, подлежащий прокрутке

Выбор вариантов

Теперь вы знаете, как вводить текст. Однако во многих случаях предпочтительнее ограничить действия пользователя выбором среди конечного набора возможностей. Эти возможности могут быть представлены кнопками или списком пунктов. (Во многих случаях такой подход освободит вас от необходимости отслеживать ошибки.) В этом разделе описаны флажки опций, переключатели, списки и линейные регуляторы.

Флажки опций

Если данные сводятся к ответу "да" или "нет", используется флажок опции. Эти флажки автоматически сопровождаются метками, позволяющими идентифицировать их. Чтобы установить флажок, нужно щелкнуть мышью на компоненте, а для

того чтобы сбросить флажок — щелкнуть повторно. Установить или сбросить флажок можно также с помощью клавиши пробела, нажав ее в момент, когда компонент обладает фокусом ввода.

На рис. 9.15 показано простое окно программы с двумя флажками опций, один из которых включает и отключает курсив, а другой — полужирный шрифт. Обратите внимание на то, что первый флажок обладает фокусом ввода. Об этом говорит прямоугольная рамка вокруг его метки. Каждый раз, когда пользователь щелкает на флажке, содержимое окна обновляется с учетом нового атрибута шрифта.

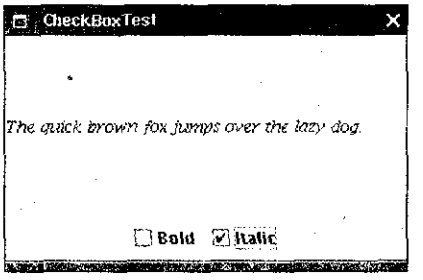


Рис. 9.15. Флажки опций

Флажки опций сопровождаются метками, указывающими их предназначение. Текст метки задается в конструкторе.

```
bold = new JCheckBox("Bold");
```

Для установки и сброса флажка используется метод `setSelected()`.

```
bold.setSelected(true);
```

Метод `isSelected()` позволяет определить текущее состояние каждого флажка. Если значение равно `false`, значит, он сброшен.

Щелкая на флажке, пользователь инициирует возникновение событий. Как всегда, вы можете связать с данным компонентом объект-обработчик. В нашей программе для двух кнопок предусмотрен один и тот же обработчик.

```
ActionListener listener = ...
bold.addActionListener(listener);
italic.addActionListener(listener);
```

В методе `actionPerformed()` обработчика запрашивается текущее состояние флажков `bold` и `italic`, а затем устанавливается шрифт, которым должен отображаться обычный текст: полужирный, курсив или полужирный курсив.

```
public void actionPerformed(ActionEvent event)
{
   int mode = 0;
   if (bold.isSelected()) mode += Font.BOLD;
   if (italic.isSelected()) mode += Font.ITALIC;
   label.setFont(new Font("Serif", mode, FONTSIZE));
}
```

В листинге 9.5 приведен полный текст программы, демонстрирующей работу с флажками опций.

Листинг 9.5. Содержимое файла `CheckBoxTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CheckBoxTest
{
   public static void main(String[] args)
   {
      CheckBoxFrame frame = new CheckBoxFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с текстовой меткой и флажками для
   установки атрибутов шрифта.
*/
class CheckBoxFrame extends JFrame
{
   public CheckBoxFrame()
   {
      setTitle("CheckBoxTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление текстовой метки.

      label = new JLabel(
         "The quick brown fox jumps over the lazy dog.");
      label.setFont(new Font("Serif", Font.PLAIN, FONTSIZE));
      add(label, BorderLayout.CENTER);

      // Данный обработчик устанавливает атрибуты шрифта,
      // которым отображается текст метки в соответствии
      // с состоянием флажков опций.

      ActionListener listener = new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               int mode = 0;
               if (bold.isSelected()) mode += Font.BOLD;
               if (italic.isSelected()) mode += Font.ITALIC;
               label.setFont(new Font("Serif", mode, FONTSIZE));
            }
         };

      // Добавление флажков опций.

      JPanel buttonPanel = new JPanel();

      bold = new JCheckBox("Bold");
      bold.addActionListener(listener);
      buttonPanel.add(bold);
```

```
        italic = new JCheckBox("Italic");
        italic.addActionListener(listener);
        buttonPanel.add(italic);

        add(buttonPanel, BorderLayout.SOUTH);
    }

    public static final int DEFAULT_WIDTH = 300;
    public static final int DEFAULT_HEIGHT = 200;

    private JLabel label;
    private JCheckBox bold;
    private JCheckBox italic;

    private static final int FONTSIZE = 12;
}
```

API **javax.swing.JCheckBox 1.2**

- `JCheckBox(String label)`

  Создает флажок опции, снабженный указанной меткой. Первоначально флажок сброшен.

- `JCheckBox(String label, bodean state)`

  Создает флажок опции, снабженный заданной меткой и находящийся в указанном состоянии.

- `JCheckBox(String label, Icon icon)`

  Создает флажок с указанной меткой и пиктограммой. Первоначально флажок сброшен.

- `boolean isSelected()`

  Возвращает состояние флажка опции.

- `void setSelected(boolean state)`

  Устанавливает новое состояние флажка опции.

Переключатели

В предыдущем примере пользователь мог установить оба флажка, один из них или ни одного. Во многих ситуациях необходимо выбрать только один из предложенных вариантов. Если пользователь установит другой флажок, то предыдущий флажок будет сброшен. Такую группу флажков часто называют *группой переключателей* (radio button group). Эти кнопки напоминают переключатели диапазонов на радиоприемниках — при нажатии одной из таких кнопок ранее нажатая кнопка возвращается в исходное состояние. На рис. 9.16 показан типичный пример окна программы, содержащей группу переключателей. Пользователь может выбрать размер шрифта — Small, Medium, Large и Extra large. Разумеется, выбрать можно лишь один размер.

Рис. 9.16. Группа переключателей

Пакет Swing позволяет легко реализовать группы переключателей. Для этого нужно создать для каждой группы по одному объекту `ButtonGroup`. Затем в группу переключателей нужно добавить объекты типа `JRadioButton`. Объект класса `ButtonGroup` предназначен для того, чтобы отключать ранее выбранный переключатель, если пользователь щелкнул на новой кнопке.

```
ButtonGroup group = new ButtonGroup();

JRadioButton smallButton
   = new JRadioButton("Small", false);
group.add(smallButton);

JRadioButton smallButton
   = new JRadioButton("Medium", true);
group.add(mediumButton);
...
```

Второй параметр конструктора принимает значение `true`, если изначально переключатель должен быть включен, и `false` — если выключен. Отметим, что объект типа `ButtonGroup` управляет лишь *поведением* переключателей. Если нужно объединить несколько групп переключателей, их следует поместить в контейнер, например в объект класса `JPanel`.

Вернемся к рис. 9.15 и 9.16. Обратите внимание на то, что переключатели отличаются по внешнему виду от флажков опций. Флажки опций изображаются в виде квадратов, причем для установленных флажков внутри имеется галочка, в то время как переключатели имеют круглую форму: включенные — с точкой внутри, а выключенные — без обозначений.

Механизм извещения о наступлении событий, связанных с переключателями, точно такой же, как и для любых других кнопок. Если пользователь включил переключатель, соответствующий объект генерирует событие. В рассматриваемом здесь примере установлен обработчик, задающий конкретный размер шрифта.

```
ActionListener listener = new
   ActionListener()
   {
      public void actionPerformed(ActionListener evt)
      {
         // Размер шрифта задается последним параметром
         // метода addRadioButton.
         label.setFont(new Font("Serif", Font.PLAIN, size));
      }
   };
```

Сравните этот обработчик с обработчиком для флажка опций. Каждому переключателю соответствует свой объект обработчика. Каждый объект обработчика точно знает, что нужно делать — установить конкретный размер шрифта. Для флажков ситуация иная. Оба флажка в нашем примере были связаны с одним и тем же обработчиком. Он вызывал метод, определяющий текущее состояние обоих флажков.

Можно ли реализовать такой подход для переключателей? Зададим один обработчик, устанавливающий конкретный размер шрифта.

```
if (smallButton.isSelected()) size = 8;
else is (mediumButton.isSelected()) size = 12;
...
```

Однако предпочтительнее использовать отдельные объекты-обработчики, поскольку они более явно связывают размер шрифта с конкретным переключателем.

В группе может быть выбран только один переключатель. Хорошо бы сразу знать, какой именно, не проверяя каждый переключатель в группе. Поскольку объект класса `ButtonGroup` управляет всеми переключателями, было бы удобно, если бы он давал нам ссылку на выбранный переключатель. Действительно, в классе `ButtonGroup` есть метод `getSelection()`, но этот метод не возвращает ссылку на выбранный переключатель. Вместо этого он возвращает ссылку на объект класса `ButtonModel` — модель, связанную с этим переключателем. К сожалению, все методы класса `ButtonModel` не представляют собой ничего ценного. Интерфейс `ButtonModel` наследует метод `getSelectedObjects()` из интерфейса `ItemSelectable`, возвращающий совершенно бесполезную ссылку `null`. Метод `getActionCommand()` выглядит предпочтительнее, поскольку позволяет определить текстовую строку, связанную с переключателем. Однако команда в модели этого переключателя снова представляет собой нулевую ссылку `null`. Только если программист явно задаст команды для каждого переключателя с помощью метода `setActionCommand()`, установив значение, соответствующее каждой команде, впоследствии можно будет определить команду включенного переключателя с помощью метода `buttonGroup.getSelection()getActionCommand()`.

В листинге 9.6 представлен полный текст программы, позволяющей установить размер шрифта с помощью переключателей.

Листинг 9.6. Содержимое файла `RadioButtonTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RadioButtonTest
{
   public static void main(String[] args)
   {
      RadioButtonFrame frame = new RadioButtonFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
```

```
   Фрейм с текстовой меткой и переключателями,
   предназначенными для выбора размера шрифта.
*/
class RadioButtonFrame extends JFrame
{
   public RadioButtonFrame()
   {
      setTitle("RadioButtonTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление метки.

      label = new JLabel(
         "The quick brown fox jumps over the lazy dog.");
      label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
      add(label, BorderLayout.CENTER);

      // Добавление переключателей.

      buttonPanel = new JPanel();
      group = new ButtonGroup();

      addRadioButton("Small", 8);
      addRadioButton("Medium", 12);
      addRadioButton("Large", 18);
      addRadioButton("Extra large", 36);

      add(buttonPanel, BorderLayout.SOUTH);
   }

   /**
      Добавление переключателей, устанавливающих размер шрифта
      для отображения текстовой метки.
      @param name Строка, которой помечается кнопка
      @param size Размер шрифта для кнопки
   */
   public void addRadioButton(String name, final int size)
   {
      boolean selected = size == DEFAULT_SIZE;
      JRadioButton button = new JRadioButton(name, selected);
      group.add(button);
      buttonPanel.add(button);

      // Обработчик, устанавливающий размер шрифта для метки.

      ActionListener listener = new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               // Размер соответствует последнему параметру
               // метода addRadioButton()
               label.setFont(new Font("Serif", Font.PLAIN, size));
            }
         };
```

```
        button.addActionListener(listener);
    }

    public static final int DEFAULT_WIDTH = 400;
    public static final int DEFAULT_HEIGHT = 200;

    private JPanel buttonPanel;
    private ButtonGroup group;
    private JLabel label;

    private static final int DEFAULT_SIZE = 12;
}
```

javax.swing.JRadioButton 1.2

- `JRadioButton(String label, boolean state)`

  Создает кнопку переключателя, помеченную заданной строкой и находящуюся в указанном состоянии.

- `JRadioButton(String label, Icon icon)`

  Создает кнопку переключателя с заданной строкой и пиктограммой. В исходном состоянии кнопка не выбрана.

javax.swing.ButtonGroup 1.2

- `void add(AbstractButton b)`

  Добавляет переключатель к группе.

- `ButtonModel getSelection()`

  Возвращает модель выбранной кнопки.

javax.swing.ButtonModel 1.2

- `String getActionCommand(String s)`

  Возвращает команду для модели кнопки.

javax.swing.AbstractButton 1.2

- `void setActionCommand(String s)`

  Задает команду для данной кнопки и ее модели.

Рамки

Если в одном окне расположено несколько групп переключателей, нужно иметь возможность различать их между собой. Для этого в библиотеке Swing предусмотрен набор *рамок*. Рамку можно задать для каждого компонента, расширяющего класс `JComponent`. Обычно рамка окружает панель, заполненную ее элементами пользовательского интерфейса, например переключателями.

Выбор рамок невелик, и все они задаются с помощью одинаковых действий.

1. Вызывается статический метод класса `BorderFactory`, создающий рамку, для которой предусмотрены следующие стили (рис. 9.17).
   - Lowered bevel (утопленная фаска).
   - Raised bevel (приподнятая фаска).
   - Etched (гравировка).
   - Line (линия).
   - Matte (*матовая*).
   - Empty (пустая – создается пустое пространство, окружающее компонент).
2. При желании с помощью метода `BorderFactory.createBorder()` добавляется заголовок.
3. По мере необходимости несколько рамок можно объединить в одну, воспользовавшись методом `BorderFactory.createCompoundBorder()`.
4. Получившаяся рамка добавляется к компоненту с помощью метода `setBorder()` класса `JComponent`.

Ниже приведен фрагмент кода, предназначенный для создания рамки, имитирующей гравировку.

```
Border etched = BorderFactory.createEtchedBorder()
Border titled = BorderFactory.createTitledBorder(etched,
   "A Title");
panel.setBorder(titled);
```

Запустите программу, код которой приведен в листинге 9.7, и вы увидите, как выглядят рамки, выполненные в разных стилях.

Разные рамки имеют разные возможности для задания ширины и цвета. Детали описаны в документации по API. Истинные энтузиасты использования рамок обрадуются, узнав, что существуют классы `SoftBevelBorder` и `LineBorder`, позволяющие сглаживать углы. Такие рамки можно создать только с помощью конструкторов этих классов – для них не существует методов `BorderFactory`.

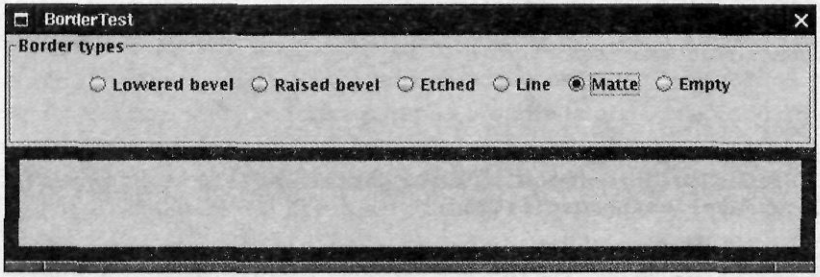


Рис. 9.17. Тестирование рамок

Листинг 9.7. Содержимое файла `BorderTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class BorderTest
{
   public static void main(String[] args)
   {
      BorderFrame frame = new BorderFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с переключателями, позволяющими выбрать стиль рамки.
*/
class BorderFrame extends JFrame
{
   public BorderFrame()
   {
      setTitle("BorderTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      demoPanel = new JPanel();
      buttonPanel = new JPanel();
      group = new ButtonGroup();

      addRadioButton("Lowered bevel",
         BorderFactory.createLoweredBevelBorder());
      addRadioButton("Raised bevel",
         BorderFactory.createRaisedBevelBorder());
      addRadioButton("Etched",
         BorderFactory.createEtchedBorder());
      addRadioButton("Line",
         BorderFactory.createLineBorder(Color.BLUE));
      addRadioButton("Matte",
         BorderFactory.createMatteBorder(10, 10, 10, 10,
         Color.BLUE));
      addRadioButton("Empty",
         BorderFactory.createEmptyBorder());

      Border etched = BorderFactory.createEtchedBorder();
      Border titled =
         BorderFactory.createTitledBorder(etched, "Border types");
      buttonPanel.setBorder(titled);

      setLayout(new GridLayout(2, 1));
      add(buttonPanel);
      add(demoPanel);
   }

   public void addRadioButton(String buttonName, final Border b)
   {
```

```
      JRadioButton button = new JRadioButton(buttonName);
      button.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               demoPanel.setBorder(b);
               validate();
            }
         });
      group.add(button);
      buttonPanel.add(button);
   }

   public static final int DEFAULT_WIDTH = 600;
   public static final int DEFAULT_HEIGHT = 200;

   private JPanel demoPanel;
   private JPanel buttonPanel;
   private ButtonGroup group;
}
```

API **javax.swing.BorderFactory 1.2**

- `static Border createLineBorder(Color color)`
- `static Border createLineBorder(Color color, int thickness)`

Создает простую рамку в виде линии.

- `static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Color color)`
- `static MatteBorder createMatteBorder(int top, int left, int bottom, int right, Icon tileIcon)`

Создает широкую рамку, заполненную цветом или мозаикой, состоящей из повторяющихся пиктограмм.

- `static Border createEmptyBorder()`
- `static Border createEmptyBorder(int top, int left, int bottom, int right)`

Создает пустую рамку.

- `static Border createEtchedBorder()`
- `static Border createEtchedBorder(Color highlight, Color shadow)`
- `static Border createEtchedBorder(int type)`
- `static Border createEtchedBorder(int type, Color highlight, Color shadow)`

Создает рамку, имитирующую трехмерное изображение.

Параметры	`highlight`, `shadow`	Цвета для создания эффекта трехмерного изображения
	`type`	Одна из констант: `EtchedBorder.RAISED` или `Etched.Border.LOWERED`

- `static Border createBevelBorder(int type)`
- `static Border createBevelBorder(int type, Color highlight, Color shadow)`
- `static Border createLoveredBevelBorder()`
- `static Border createRaisedBevelBorder()`

Создает рамку, имитирующую утопленную или приподнятую поверхность.

Параметры	`type`	Одна из констант: `BevelBorder.RAISED` или `BevelBorder.LOWERED`
	`highlight, shadow`	Цвета для создания эффекта трехмерного изображения

- `static TitledBorder createTitledBorder(String title)`
- `static TitledBorder createTitledBorder(Border border)`
- `static TitleBorder createTitleBorder(Border Border, String title)`
- `static TitleBorder createTitleBorder(Border Border, String title, int justification, int position)`
- `static TitleBorder createTitleBorder(Border Border, String title, int justification, int position, Font font)`
- `static TitleBorder createTitleBorder(Border border, String title, int justification, int position, Font font, Color color)`

Создает рамку с заданными свойствами, снабженную заголовком.

Параметры	`title`	Строка заголовка
	`border`	Рамка, снабженная заголовком
	`justification`	Одна из констант: `LEFT`, `CENTER`, `RIGHT`, `LEADING`, `TRAILING` или `DEFAULT_JUSTIFICATION` (аналогично `LEFT`)
	`position`	Одна из констант класса `TitleBorder`: `ABOVE_TOP`, `TOP`, `BELOW_TOP`, `ABOVE_BOTTOM`, `BOTTOM`, `BELOW_BOTTOM` или `DEFAULT_POSITION` (аналогично `TOP`)
	`font`	Шрифт заголовка
	`color`	Цвет заголовка

- `static CompoundBorder createCompoundBorder (Border outsideBorder, Border inside Border)`

Объединяет две рамки в одно целое.

API **`javax.swing.border.SoftBevelBorder 1.2`**

- `SoftBevelBorder(int type)`
- `SoftBevelBorder(int type, Color highlight, color shadow)`

Создает фаску со сглаженными углами.

Параметры	`type`	Одна из констант: `BevelBorder.RAISED` или `BevelBorder.LOWERED`
	`highlight, shadow`	Цвета для имитации трехмерного изображения

`javax.swing.border.LineBorder 1.2`

- `public LineBorder(Color color, int thickness, boolean roundedCorners)`

  Создает рамку в виде линии заданной ширины и цвета. Если значение параметра `roundedCorners` равно `true`, рамка имеет скругленные углы.

`javax.swing.JComponent 1.2`

- `void setBorder(Border border)`

  Задает рамку для компонента.

Раскрывающиеся списки

Если вариантов для выбора слишком много, переключатели не подойдут, поскольку для них не хватит места на экране. В этом случае следует использовать раскрывающиеся списки. Если пользователь щелкнет на этом компоненте, откроется список из нескольких пунктов и пользователь сможет выбрать один из них (рис. 9.18).

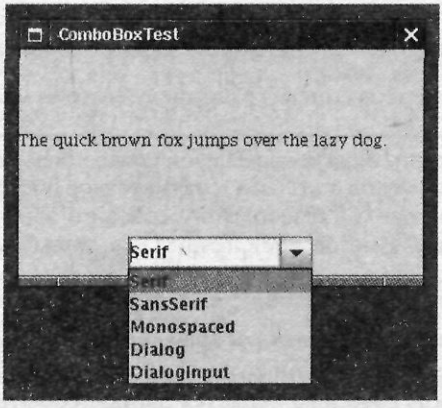


Рис. 9.18. Раскрывающийся список

Если раскрывающийся список является *редактируемым* (editable), то строку, соответствующую выбранному пункту, *можно редактировать* как содержимое обычного текстового поля. Таким образом, редактируемый раскрывающийся список объединяет в себе гибкость текстового поля и возможность выбора из фиксированного набора возможностей, предусмотренных заранее. Раскрывающиеся списки реализуются с помощью класса `JComboBox`.

Для того чтобы сделать раскрывающийся список редактируемым, следует вызвать метод `setEditable()`. Обратите внимание на то, что изменения вносятся только в текущий пункт. Остальная часть списка остается неизменной.

Выбранную опцию или редактируемый текст можно получить с помощью метода `getSelectedItem()`.

В программе, рассматриваемой здесь в качестве примера, пользователь может выбрать стиль шрифта из заданного заранее списка (Serif, SunsSerif, Monospaced и т.д.). Кроме того, пользователь может внести в список новый шрифт.

Добавить пункт в список можно с помощью метода `addItem()`. В нашей программе метод `addItem()` вызывается только в конструкторе, но при необходимости к нему можно обратиться из любого метода.

```
faceCombo = new JComboBox();
faceCombo.setEditable(true);
faceCombo.addItem("Serif");
faceCombo.addItem("SansSerif");
...
```

Этот метод добавляет строку в конец списка. Если строку нужно вставить в другую позицию, следует вызвать метод `insertItemAdd()`.

```
faceCombo.insertItemAt("Monospaced", 0); //Вставка в начало списка.
```

В список можно включать элементы любого типа — для отображения их будет вызван метод `toString()`.

Если во время выполнения программы возникает необходимость удалить пункт списка, используются методы `removeItem()` или `removeItemAt()`, в зависимости от того, хотите ли вы удалить текущий элемент или тот, который находится в указанной позиции списка.

```
faceCombo.removeItem("Monospaced");
faceCombo.removeItemAt(0); // Удаляется первый элемент.
```

Для удаления всех элементов списка сразу предусмотрен метод `removeAllItems()`.

Если вам надо включить в раскрывающийся список большое количество объектов, не следует использовать метод `addItem()`, так как такой подход существенно снизит производительность программы. Вместо этого создайте объект `DefaultComboBoxModel`, заполните его, вызывая метод `addElement()`, а затем вызовите метод `setModel()` класса `JComboBox`.

Когда пользователь выбирает пункт раскрывающегося списка, компонент генерирует событие. Чтобы определить, какой пункт списка был выбран, надо вызвать метод `getSource()`, параметром которого является данное событие. Он возвращает ссылку на список, являющийся источником события. Затем следует вызвать метод `getSelectedItem()`, возвращающий выбранный пункт списка. Значение, возвращенное этим методом, нужно привести к соответствующему типу, обычно к типу `String`.

```
public void actionPerformed(ActionEvent event);
{
   label.setFont(new Font(
      (String)faceCombo.getSelectedtem(),
      Font.Plain,
      DEFAULT_SIZE));
}
```

Полный текст программы приведен в листинге 9.8.

Если вы хотите, чтобы вместо раскрывающегося отображался обычный список, т.е. чтобы пункты списка постоянно отображались на экране, используйте компонент `JList`. Мы рассмотрим его в главе 6 второго тома.

Листинг 9.8. Содержимое файла `ComboBoxTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ComboBoxTest
{
   public static void main(String[] args)
   {
      ComboBoxFrame frame = new ComboBoxFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с текстовой меткой и раскрывающимся
   списком, предназначенным для выбора начертаний.

*/
class ComboBoxFrame extends JFrame
{
   public ComboBoxFrame()
   {
      setTitle("ComboBoxTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление текстовой метки.

      label = new JLabel(
         "The quick brown fox jumps over the lazy dog.");
      label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
      add(label, BorderLayout.CENTER);

      // Создание раскрывающегося списка и включение
      // в него пунктов, соответствующих начертаниям.

      faceCombo = new JComboBox();
      faceCombo.setEditable(true);
      faceCombo.addItem("Serif");
      faceCombo.addItem("SansSerif");
      faceCombo.addItem("Monospaced");
      faceCombo.addItem("Dialog");
      faceCombo.addItem("DialogInput");

      // Обработчик событий, связанных с раскрывающимся
      // списком, изменяет шрифт, которым отображается
      // метка.
```

```
      faceCombo.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               label.setFont(new Font(
                  (String) faceCombo.getSelectedItem(),
                  Font.PLAIN,
                  DEFAULT_SIZE));
            }
         });

      // Включение раскрывающегося списка в состав панели
      // (панель располагается в нижней части фрейма).

      JPanel comboPanel = new JPanel();
      comboPanel.add(faceCombo);
      add(comboPanel, BorderLayout.SOUTH);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   private JComboBox faceCombo;
   private JLabel label;
   private static final int DEFAULT_SIZE = 12;
}
```

API javax.swing.JComboBox 1.2

- `void setEditable(boolean b)`

  Параметр `b` Значение `true`, если пункт списка можно редактировать, и `false` — если редактирование запрещено

- `void addItem(Object item)`

  Включает в список новый элемент.

- `void addItem(Object item)`

  Добавляет элемент в заданную позицию списка.

- `void removeItem(Object item)`

  Удаляет заданный элемент из списка.

- `void removeItem(int index)`

  Удаляет из списка элемент с заданным номером.

- `void removeAllItems()`

  Удаляет все элементы из списка.

- `Object getSelectedItem()`

  Возвращает выбранный элемент списка.

Линейные регуляторы

Раскрывающиеся списки позволяют пользователю делать выбор из дискретного набора значений. Линейные регуляторы дают возможность выбрать конкретное значение из континуума возможных, например любое число из интервала 1–100.

Наиболее часто линейные регуляторы создаются следующим образом:

```
JSlider slider = new JSlider(min, max, initialValue);
```

Если не указывать минимальное, максимальное и начальное значение, принимаются соответственно величины 0, 100 и 50.

Если линейные регулятор должен располагаться вертикально, используется приведенный ниже конструктор.

```
JSlider slider = new JSlider(SwingConstants.VERTICAL,
   min, max, initialValue);
```

Каждый из этих конструкторов создает простой регулятор. В качестве примера можно привести регулятор, который расположен в верхней части окна, представленного на рис. 9.19. Позже мы рассмотрим более сложные варианты линейных регуляторов.

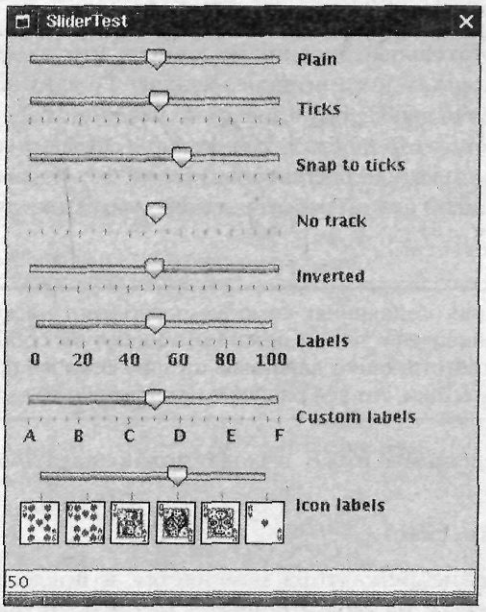


Рис. 9.19. Линейные регуляторы

Когда пользователь перемещает ползунок регулятора, *значение* компонента изменяется в пределах от минимального до максимального. Все обработчики, связанные с линейным регулятором, получают событие `ChangeEvent`. Для того чтобы получать извещения об изменении значения регулятора, нужно создать объект, реализующий интерфейс `ChangeListener`, и вызвать метод `addChangeListener()`. В этом интерфейсе объявлен лишь один метод `stateChanged()`. При его выполнении определяется значение регулятора.

```
public void stateChanged(ChangeEvent event)
{
   JSlider slider = (JSlider)event.getSource();
   int value = slider.getValue();
   ...
}
```

Линейный регулятор можно дополнить *делениями* (tic). Например, в программе, рассматриваемой в качестве примера, для второго регулятора производятся следующие установки:

```
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
```

Регулятор снабжен основными делениями, следующими через каждые 20 единиц измерения, и вспомогательными, которые следуют через каждые 5 единиц. В данном случае единицы измерения связываются со значениями регулятора и не имеют никакого отношения к пикселям.

Приведенные выше выражения лишь устанавливают деления регулятора. Чтобы вывести их на экран, нужно вызвать следующий метод:

```
slider.setpaintTicks(true);
```

Основные и вспомогательные деления не зависят друг от друга. Например, можно задать расстановку основных делений через каждые 20 единиц, а вспомогательных — через 7 единиц, хотя такая шкала будет довольно неудобной.

Можно перевести регулятор в режим *фиксации на делениях* (snap to ticks). Как только пользователь отпустит ползунок регулятора, он тут же переместится на ближайшее деление. Чтобы задать такой режим работы, нужно вызвать следующий метод:

```
slider.setSnapToTicks(true);
```

В режиме фиксации на делениях компонент работает несколько неожиданно. Пока регулятор не установится точно на деление, обработчик событий изменений получает значения, не соответствующие делениям шкалы. Если же пользователь перемещает ползунок щелчком мыши, тот все равно не попадает на ближайшее деление.

С помощью приведенного ниже вызова можно задать *метки основных делений* (tick mark labels).

```
slider.setPaintLabels(true);
```

Например, если значение регулятора изменяется в пределах от 0 до 100, а шаг основных делений равен 20, на экране появятся метки 0, 20, 40, 60, 80 и 100.

Можно задать и другие типы меток, например строки или пиктограммы (см. рис. 9.19). Эта задача не тривиальна. Сначала нужно заполнить хэш-таблицу с ключами типа `Integer` и значениями типа `Component` (автоматическое преобразование, реализованное в JDK 5.0, упрощает этот процесс). Затем следует вызвать метод `setLabelTable()`. Компоненты размещаются под делениями. Обычно используются объекты класса `JLabel`. Ниже показано, как можно задать метки делений A, B, C, D, E и F.

```
Hashtable<Integer, Component> labelTable =
   new Hashtable<Integer, Component>();
labelTable.put(0, new JLabel("A"));
```

```
labelTable.put(20, new JLabel("B"));
. . .
labelTable.put(100, new JLabel("F"));
slider.setLabelTable(labelTable);
```

Более подробно хэш-таблицы описаны в главе 2 тома 2.

В листинге 9.9 показан пример создания регулятора, у которого метками деления являются пиктограммы.

Если деления или их метки не выводятся на экран, проверьте, вызваны ли методы `setPaintTicks(true)` и `setpaintLabels(true)`.

Подавить изображение "дорожки", по которой двигается ползунок, можно с помощью вызова

```
slider.setPaintTrack(false);
```

В результате регулятор будет выглядеть так, как четвертый сверху элемент на рис. 9.19.

Для пятого регулятора на этом рисунке направление движение изменено посредством следующего метода:

```
slider.setInverted(true);
```

Набор регуляторов, создаваемых в приведенном здесь примере, демонстрирует различные визуальные эффекты. Для каждого регулятора установлен обработчик событий, который отображает текущее значение выбранного регулятора в поле редактирования, расположенном в нижней части окна.

Листинг 9.9. Содержимое файла `SliderTest.java`

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

public class SliderTest
{
   public static void main(String[] args)
   {
      SliderTestFrame frame = new SliderTestFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с линейными регуляторами и полем редактирования,
   отображающим текущее значение выбранного регулятора.
*/
class SliderTestFrame extends JFrame
{
   public SliderTestFrame()
   {
```

```
setTitle("SliderTest");
setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

sliderPanel = new JPanel();
sliderPanel.setLayout(new FlowLayout(FlowLayout.LEFT));

// Общий обработчик для всех регуляторов.
listener = new
   ChangeListener()
   {
      public void stateChanged(ChangeEvent event)
      {
         // Обновление поля редактирования при
         // изменении значения регулятора.
         JSlider source = (JSlider) event.getSource();
         textField.setText("" + source.getValue());
      }
   };

// Добавление обычного регулятора.

JSlider slider = new JSlider();
addSlider(slider, "Plain");

// Добавление регулятора с основными и
// вспомогательными делениями.

slider = new JSlider();
slider.setPaintTicks(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
addSlider(slider, "Ticks");

// Добавление регулятора с фиксацией на делениях.

slider = new JSlider();
slider.setPaintTicks(true);
slider.setSnapToTicks(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
addSlider(slider, "Snap to ticks");

// Добавление регулятора, для которого "дорожка",
// по которой перемещается ползунок, не отображается.

slider = new JSlider();
slider.setPaintTicks(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
slider.setPaintTrack(false);
addSlider(slider, "No track");

// Добавление регулятора с обратным направлением движения.

slider = new JSlider();
slider.setPaintTicks(true);
```

```
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
slider.setInverted(true);
addSlider(slider, "Inverted");

// Добавление регулятора с числовыми метками.

slider = new JSlider();
slider.setPaintTicks(true);
slider.setPaintLabels(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
addSlider(slider, "Labels");

// Добавление регулятора с буквенными метками.

slider = new JSlider();
slider.setPaintLabels(true);
slider.setPaintTicks(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);

Dictionary<Integer, Component> labelTable =
   new Hashtable<Integer, Component>();
labelTable.put(0, new JLabel("A"));
labelTable.put(20, new JLabel("B"));
labelTable.put(40, new JLabel("C"));
labelTable.put(60, new JLabel("D"));
labelTable.put(80, new JLabel("E"));
labelTable.put(100, new JLabel("F"));

slider.setLabelTable(labelTable);
addSlider(slider, "Custom labels");

// Добавление регулятора с метками в виде пиктограмм.

slider = new JSlider();
slider.setPaintTicks(true);
slider.setPaintLabels(true);
slider.setSnapToTicks(true);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(20);

labelTable = new Hashtable<Integer, Component>();

// Добавление изображений.

labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));
labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));
labelTable.put(40, new JLabel(new ImageIcon("jack.gif")));
labelTable.put(60, new JLabel(new ImageIcon("queen.gif")));
labelTable.put(80, new JLabel(new ImageIcon("king.gif")));
labelTable.put(100, new JLabel(new ImageIcon("ace.gif")));

slider.setLabelTable(labelTable);
addSlider(slider, "Icon labels");
```

```
      // Добавление поля редактирования, предназначенного
      // для отображения значения регулятора.

      textField = new JTextField();
      add(sliderPanel, BorderLayout.CENTER);
      add(textField, BorderLayout.SOUTH);
   }

   /**
      Добавление регулятора на панель и связывание его с
      обработчиком.
      @param s Регулятор
      @param description Описание регулятора
   */
   public void addSlider(JSlider s, String description)
   {
      s.addChangeListener(listener);
      JPanel panel = new JPanel();
      panel.add(s);
      panel.add(new JLabel(description));
      sliderPanel.add(panel);
   }

   public static final int DEFAULT_WIDTH = 350;
   public static final int DEFAULT_HEIGHT = 450;

   private JPanel sliderPanel;
   private JTextField textField;
   private ChangeListener listener;
}
```

API **javax.swing.JSlider 1.3**

- `JSlider()`
- `JSlider(int direction)`
- `JSlider(int min, int max)`
- `JSlider(int min, int max, ini initialValue)`
- `JSlider(int direction, int min, int max, ini initialValue)`

Создает линейный регулятор с заданным направлением, минимальным и максимальным значениями.

Параметры	`direction`	Одна из констант: `SwingConstantc.HORIZONTAL` или `swingConstants.VERTICAL`. По умолчанию принимается значение `SwingConstantc.HORIZONTAL`
	`min, max`	Минимальное и максимальное значения регулятора. По умолчанию равны 0 и 100
	`initialValue`	Начальное значение регулятора. По умолчанию равно 50

- `void setPaintTicks(boolean b)`

  Если значение параметра b равно `true`, деления отображаются на экране.

- `void setMajorTickSpacing(int units)`
- `void setMinorTickSpacing(int units)`

  Устанавливают основную и вспомогательную шкалы.

- `void setPaintLabels(boolean b)`

  Если значение параметра b равно `true`, деления отображаются на экране.

- `slider.setlabelTable(Dictionary table)`

  Устанавливает компоненты, которые используются в качестве меток делений. Каждая пара ключ–значение представлена в виде `new Integer` (*значение*) / *компонент*.

- `void setSnapToTicks(boolean b)`

  Если значение параметра b равно `true`, ползунок регулятора всегда перемещается на деление, ближайшее к установленному значению регулятора.

- `void setPaintTrack(boolean b)`

  Если значение параметра равно `true`, на экране отображается "дорожка", по которой перемещается ползунок.

Инкрементный регулятор

Компонент `JSpinner` представляет собой поле редактирования с двумя небольшими кнопками, размещенными сбоку и предназначенными для увеличения или уменьшения значения, хранящегося в поле (рис. 9.20).

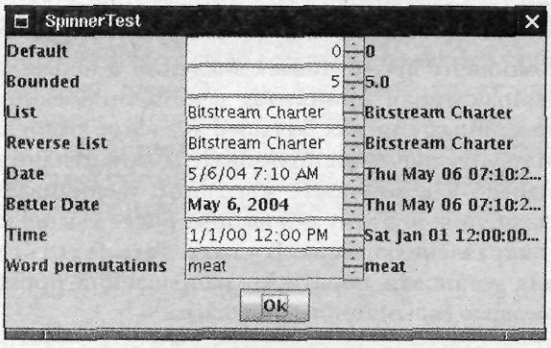


Рис. 9.20. Разновидности компонента `JSpinner`

Значениями этого компонента могут быть числа, даты и элементы списка, т.е. любая последовательность величин, в которой можно определить порядок их следования. В классе `JSpinner` определены стандартные способы представления чисел, дат и элементов списка. Для описания произвольной последовательности значений пользователь может определить свою собственную модель.

По умолчанию инкрементный регулятор управляет целыми числами, а кнопки увеличивают или уменьшают эти числа на единицу. Для получения текущего значения следует вызвать метод `getValue()`. Этот метод возвращает объект класса `Object`. Его следует привести к типу `Integer`, а затем извлечь значение.

```
JSpinner defaultSpinner = new JSpinner();
. . .
int value = (Integer) defaultSpinner.getValue();
```

Приращение значения можно изменять. Кроме того, можно задать нижнюю и верхнюю границы изменения значений. Рассмотрим компонент `JSpinner`, у которого исходное значение равно 5, нижняя и верхняя границы равны соответственно 0 и 10, а приращение равно 0,5.

```
JSpinner boundedSpinner = new
   JSpinner(new SpinnerNumberModel(5, 0, 10, 0.5));
```

Существуют два конструктора `SpinnerNumberModel`: у одного из них все параметры имеют тип `int`, а у другого — `double`. Если какой-либо из параметров представляет собой число с плавающей точкой, вызывается второй конструктор. Он задает в качестве значения компонента `JSpinner` объект класса `Double`.

Компоненты `JSpinner` не ограничены лишь числами. Они могут обрабатывать любой набор значений. Для этого достаточно передать объект класса `SpinnerListModel` конструктору класса `JSpinner`. Объект класса `SpinnerListModel` можно создать из массива или класса, реализующего интерфейс `List` (например, класса `ArrayList`). В программе, которая будет рассмотрена в качестве примера, компонент `JSpinner` управляет именами шрифтов.

```
String[] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment()
   .getAvailableFontFamilyNames();
JSpinner listSpinner = new JSpinner(new SpinnerListModel(fonts));
```

Оказалось, что компонент прокручивает значения в неожиданном направлении, противоположном направлению, принятому в комбинированных списках. Как вы помните, в комбинированных списках большее значение размещается *под* меньшим, поэтому для поиска больших значений пользователь интуитивно выберет стрелку, направленную вниз. Однако в компоненте `JSpinner` увеличивается индекс массива, в котором хранятся значения, поэтому для выбора более высоких значений нужно использовать стрелку, направленную вверх. В классе `SpinnerListModel` не предусмотрена возможность для установки обратного направления прокрутки, однако этого можно добиться с помощью безымянного подкласса.

```
JSpinner reverseListSpinner = new JSpinner(
   new SpinnerListModel(fonts)
   {
      public Object getNextValue()
      {
         return super.getPreviousValue();
      }
      public Object getPreviousValue()
      {
          return super.getNextValue();
      }
   });
```

Испытайте обе версии и определите, какая из них лучше подходит для ваших целей.

Инкрементные регуляторы также очень удобны для работы с датами. Например, можно создать объект класса JSpinner, инициализированный текущей датой.

```
JSpinner dateSpinner = new JSpinner(new SpinnerDateModel());
```

Однако, если внимательно рассмотреть рис. 9.20, можно увидеть, что в поле редактирования отображаются не только дата, но и время.

```
3/12/02  7:23  PM
```

Однако время нас в данном случае не интересует! Оказывается, что создать объект класса JSpinner, который отображает только дату, довольно сложно.

```
JSpinner betterDateSpinner = new JSpinner(new SpinnerDateModel());
String pattern =
   ((SimpleDateFormat) DateFormat.getDateInstance()).toPattern();
betterDateSpinner.setEditor(new
   JSpinner.DateEditor(betterDateSpinner, pattern));
```

Используя аналогичный подход, можно создать объект, управляющий только временем. Для этого нужно применить конструктор SpinnerDateModel, позволяющий задавать объект класса Date, нижнюю и верхнюю границы (или ссылку null, если границ нет) и модифицируемое поле класса Calendar (например, Calendar.HOUR).

```
JSpinner timeSpinner = new JSpinner(
   new SpinnerDateModel(
      new GregorianCalendar(2000,
         Calendar.JANUARY, 1, 12, 0, 0).getTime(),
         null, null, Calendar.HOUR));
```

Однако, если вам захочется задать 15-минутное приращение, стандартных возможностей класса SpinnerDateModel окажется недостаточно.

Для отображения произвольной последовательности значений в инкрементном регуляторе необходимо определить свою собственную модель прокрутки. В программе, рассматриваемой в качестве примера, мы заставили объект класса JSpinner перебирать все возможные перестановки букв в слове "meat": "mate", "meta", "team" и еще двадцать перестановок можно вывести на экран, щелкая на кнопках этого компонента.

Определяя свою собственную модель, необходимо расширить класс AbstractSpinnerModel и определить четыре метода:

```
Object getValue()
void SetValue(Object value)
Object getNextValue()
Object getPreviousValue()
```

Метод getValue() возвращает значение, хранящееся в модели. Метод setValue() задает новое значение. Если значение является неприемлемым, метод setValue() должен генерировать исключение IllegalArgumentException.

После установки нового значения метод setValue() должен вызывать метод fireStateChanged(). В противном случае поле редактирования компонента JSpinner останется неизменным.

Методы getNextValue() и getPreviousValue() возвращают предшествующее или предыдущее значение либо ссылку null, если достигнут конец набора.

Методы `getNextValue()` и `getPreviousValue()` *не должны* изменять текущее значение. Когда пользователь щелкает на стрелке, направленной вверх, вызывается метод `getNextValue()`. Если возвращается не `null`, значение устанавливается с помощью метода `setValue()`.

В приведенном ниже примере использован стандартный алгоритм определения следующей и предыдущей перестановки. Детали этого алгоритма не представляют для нас интереса.

В листинге 9.10 показан способ генерации различных объектов класса `JSpinner`. Для просмотра значений щелкните на кнопке OK.

Листинг 9.10. Содержимое файла `SpinnerTest.java`

```
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;
import javax.swing.*;

/**
   Программа для проверки инкрементных регуляторов.
*/
public class SpinnerTest
{
   public static void main(String[] args)
   {
      SpinnerFrame frame = new SpinnerFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с панелью, содержащей несколько инкрементных регуляторов
   и кнопку для просмотра их значений.
*/
class SpinnerFrame extends JFrame
{
   public SpinnerFrame()
   {
      setTitle("SpinnerTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      JPanel buttonPanel = new JPanel();
      okButton = new JButton("OK");
      buttonPanel.add(okButton);
      add(buttonPanel, BorderLayout.SOUTH);

      mainPanel = new JPanel();
      mainPanel.setLayout(new GridLayout(0, 3));
      add(mainPanel, BorderLayout.CENTER);

      JSpinner defaultSpinner = new JSpinner();
      addRow("Default", defaultSpinner);

      JSpinner boundedSpinner = new JSpinner(new
```

```
            SpinnerNumberModel(5, 0, 10, 0.5));
      addRow("Bounded", boundedSpinner);

      String[] fonts = GraphicsEnvironment
         .getLocalGraphicsEnvironment()
         .getAvailableFontFamilyNames();

      JSpinner listSpinner = new JSpinner(new
         SpinnerListModel(fonts));
      addRow("List", listSpinner);

      JSpinner reverseListSpinner = new JSpinner(
         new
            SpinnerListModel(fonts)
            {
               public Object getNextValue()
               {
                  return super.getPreviousValue();
               }
               public Object getPreviousValue()
               {
                  return super.getNextValue();
               }
            });
      addRow("Reverse List", reverseListSpinner);

      JSpinner dateSpinner = new JSpinner(new SpinnerDateModel());
      addRow("Date", dateSpinner);

      JSpinner betterDateSpinner = new JSpinner(new
         SpinnerDateModel());
      String pattern = ((SimpleDateFormat)
         DateFormat.getDateInstance()).toPattern();
      betterDateSpinner.setEditor(new
         JSpinner.DateEditor(betterDateSpinner, pattern));
      addRow("Better Date", betterDateSpinner);

      JSpinner timeSpinner = new JSpinner(
         new SpinnerDateModel(
            new GregorianCalendar(2000, Calendar.JANUARY,
               1, 12, 0, 0).getTime(),
               null, null, Calendar.HOUR));
      addRow("Time", timeSpinner);

      JSpinner permSpinner = new JSpinner(new
         PermutationSpinnerModel("meat"));
      addRow("Word permutations", permSpinner);
   }

   /**
      Добавление строки к главной панели.
      @param labelText Метка компонента JSpinner
      @param spinner Пример компонента JSpinner
   */
   public void addRow(String labelText, final JSpinner spinner)
   {
```

```
      mainPanel.add(new JLabel(labelText));
      mainPanel.add(spinner);
      final JLabel valueLabel = new JLabel();
      mainPanel.add(valueLabel);
      okButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               Object value = spinner.getValue();
               valueLabel.setText(value.toString());
            }
         });
   }

   public static final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 250;

   private JPanel mainPanel;
   private JButton okButton;
}

/**
   Модель, динамически генерирующая перестановки букв в слове
*/
class PermutationSpinnerModel extends AbstractSpinnerModel
{
   /**
      Конструктор модели.
      @param w Слово для перестановки
   */
   public PermutationSpinnerModel(String w)
   {
      word = w;
   }

   public Object getValue()
   {
      return word;
   }

   public void setValue(Object value)
   {
      if (!(value instanceof String))
         throw new IllegalArgumentException();
      word = (String) value;
      fireStateChanged();
   }

   public Object getNextValue()
   {
      int[] codePoints = toCodePointArray(word);
      for (int i = codePoints.length - 1; i > 0; i--)
      {
         if (codePoints[i - 1] < codePoints[i])
         {
```

```
            int j = codePoints.length - 1;
            while (codePoints[i - 1] > codePoints[j]) j--;
            swap(codePoints, i - 1, j);
            reverse(codePoints, i, codePoints.length - 1);
            return new String(codePoints, 0, codePoints.length);
         }
      }
      reverse(codePoints, 0, codePoints.length - 1);
      return new String(codePoints, 0, codePoints.length);
   }

   public Object getPreviousValue()
   {
      int[] codePoints = toCodePointArray(word);
      for (int i = codePoints.length - 1; i > 0; i--)
      {
         if (codePoints[i - 1] > codePoints[i])
         {
            int j = codePoints.length - 1;
            while (codePoints[i - 1] < codePoints[j]) j--;
            swap(codePoints, i - 1, j);
            reverse(codePoints, i, codePoints.length - 1);
            return new String(codePoints, 0, codePoints.length);
         }
      }
      reverse(codePoints, 0, codePoints.length - 1);
      return new String(codePoints, 0, codePoints.length);
   }

   private static int[] toCodePointArray(String str)
   {
      int[] codePoints = new int[str.codePointCount(0,
         str.length())];
      for (int i = 0, j = 0; i < str.length(); i++, j++)
      {
         int cp = str.codePointAt(i);
         if (Character.isSupplementaryCodePoint(cp)) i++;
         codePoints[j] = cp;
      }
      return codePoints;
   }

   private static void swap(int[] a, int i, int j)
   {
      int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }

   private static void reverse(int[] a, int i, int j)
   {
      while (i < j) { swap(a, i, j); i++; j--; }
   }

   private String word;
}
```

javax.swing.JSpinner 1.4

- `JSpinner()`

  Создает объект класса `JSpinner` для редактирования целых чисел. Начальное значение равно 0, приращение равно 1, граничные значения отсутствуют.

- `JSpinner(SpinnerModel model)`

  Создает инкрементный регулятор, использующий заданную модель данных.

- `Object getValue()`

  Определяет текущее значение инкрементного регулятора.

- `void setValue(Object value)`

  Пытается установить текущее значение инкрементного регулятора. Если значение неприемлемо, генерируется исключение `IllegalArgumentException`.

- `void setEditor(JComponent editor)`

  Задает компонент, который используется для редактирования значения инкрементного регулятора.

javax.swing.SpinnerNumberModel 1.4

- `SpinnerNumberModel(int initval, int minimum, int maximum, int stepSize)`
- `SpinnerNumberModel(double initval, double minimum, double maximum, double stepSize)`

Эти конструкторы создают числовые модели, управляющие объектами классов `Integer` или `Double`. Для неограниченных величин используются константы `MIN_VALUE` и `MAX_VALUE`, определенные в классах `Integer` и `Double`.

Параметры	`initval`	Начальное значение
	`minimum`	Минимально допустимое число
	`maximum`	Максимально допустимое число
	`stepSize`	Приращение

javax.swing.SpinnerListMode 1.4

- `SpinnerListModel(Object[] values)`
- `SpinnerListModel(List values)`

Эти конструкторы создают модели выбора значения из заданного набора.

javax.swing.SApinnerDateModel 1.4

- `SpinnerDateModel()`

  Создает модель данных, инициализированную текущей датой, не имеющую верхней и нижней границ. В качестве шага изменения значения принято `Calendar.DAY_OF_MONTH`.

- `SpinnerDateModel(Date initval, Comparable minimum, Comparable maximum, int step)`

Параметры		
	`initval`	Начальное значение
	`minimum`	Минимально допустимое значение, если не указана нижняя граница
	`maximum`	Максимально допустимое значение, если не указана верхняя граница
	`step`	Приращение даты. Может задаваться одной из следующий констант: `ERA`, `YEAR`, `MONTH`, `WEEK_OF_YEAR`, `WEEK_OF_MONTH`, `DAY_OF_MONTH`, `DAY_OF_YEAR`, `DAY_OF_WEEK`, `DAY_OF_WEEK_IN_MONTH`, `AM_PM`, `HOUR`, `HOUR_OF_DAY`, `MINUTE`, `SECOND` или `MILLISECOND`, определенных в классе `Calendar`

API `javax.text.SimpleDateFormat 1.1`

- `String toPattern()` **1.2**

Определяет шаблон редактирования, используемый в данном методе форматирования. Типичный шаблон выглядит так: `"yyyy-MM-dd"`. Более детальное описание шаблона приведено в документации по пакету JDK.

API `javax.swing.JSpinner.DateEditor 1.4`

- `DateEditor(JSpinner spinner, String pattern)`

Создает редактор данных для инкрементного регулятора.

Параметры		
	`spinner`	Объект класса `JSpinner`, которому принадлежит редактор
	`pattern`	Формальный шаблон для `SimpleDateFormat`

API `javax.swing.AbstractSpinnerModel 1.4`

- `Object getValue()`

Определяет текущее значение модели.

- `void setValue(Object value)`

Пытается задать новое значение модели. Если это значение неприемлемо, генерирует исключение `IllegalArgumentException`. Переопределяя этот метод, после задания нового значения следует вызывать метод `fireStateChanged()`.

- `Object getNextValue()`
- `Object getPreviousValue()`

Эти методы вычисляют (но не устанавливают) последующее и предыдущее значения в последовательности, определенной в данной модели.

Меню

В начале этой главы мы описали наиболее распространенные компоненты, которые можно разместить в окне, например кнопки, поля редактирования и раскрывающиеся списки. В пакете Swing предусмотрены также выпадающие меню, хорошо известные пользователям, которые когда-либо работали с программами с графическим интерфейсом.

Строка меню (menu bar) в верхней части окна содержит названия выпадающих меню. Щелкая на имени, пользователь открывает меню, состоящее из *пунктов* и *подменю*. Если пользователь щелкнет на пункте меню, все меню закроются и в программу будет передано сообщение. На рис. 9.21 показано типичное меню, содержащее пункты и подменю.

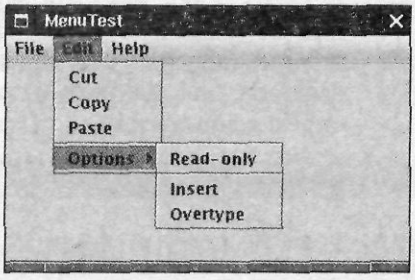


Рис. 9.21. Меню и подменю

Создание меню

Создать меню достаточно просто. Сначала создается строка меню.

```
JMenuBar menuBar = new JMenuBar();
```

Строка меню — это обычный компонент, который можно разместить где угодно. Как правило, его помещают в верхнюю часть фрейма с помощью метода `setJMenuBar()`.

```
frame.setJMenuBar(menuBar);
```

Для каждого меню создается свой объект.

```
Jmenu editMenu = new Jmenu("Edit");
```

Меню верхнего уровня помещаются в строке меню.

```
menuBar.add(editMenu);
```

Затем в объект меню добавляются пункты, разделители и подменю.

```
JMenuItem pasteItem = new JMenuItem("Paste");
editMenu.add(pasteItem);
editMenu.addSeparator();
Jmenu optionsMenu = ...; // Подменю.
editMenu.add(optionsMenu);
```

Разделители показаны на рис. 9.21. Они отображаются под пунктами Paste и Read-only.

Когда пользователь выбирает меню, генерируются события. Следовательно, для каждого пункта меню надо определить обработчик.

```
ActionListener listener = ...;
pasteItem.addActionListener(listener);
```

Существует удобный метод JMenu.add(String), позволяющий добавлять новый пункт в конец меню, например, следующим образом:

```
editMenu.add("Paste");
```

Этот метод возвращает созданный пункт меню, для которого можно легко задать обработчик.

```
JMenuItem pasteItem = editMenu.add("Paste");
pasteItem.addActionListener(listener);
```

Довольно часто пункты меню связываются с командами, которые могут активизировать другие элементы пользовательского интерфейса, например кнопки. В главе 8 говорилось о том, как задаются команды с помощью объектов Action. Следует определить класс, реализующий интерфейс Action. Обычно для этого класс расширяет AbstractAction. Затем в конструкторе AbstractAction указывается метка пункта меню и переопределяется метод actionPerformed(), что позволяет реализовать обработку события, связанного с данным пунктом.

```
Action exitAction = new
   AbstractAction("Exit") // Пункт меню "Exit".
   {
     public void actionPerformed(ActionEvent event)
     {
        // Фрагмент программы, обрабатывающий событие.
        System.exit(0);
     }
};
```

Затем объект Action добавляется в меню.

```
JMenuItem exitItem = fileMenu.add(exitAction);
```

В результате выполнения данного выражения в меню добавляется новый пункт. Объект Action становится обработчиком. Такой способ заменяет следующие строки кода:

```
JMenuItem exitItem = new JMenuItem(exitAction);
fileMenu.add(exitItem);
```

В программах для Windows и Macintosh меню обычно определяются во внешних файлах ресурсов и связываются с приложениями с помощью идентификаторов ресурсов. Меню можно создавать непосредственно в программе, однако, как правило, этого не делают. В языке Java меню создаются обычным образом, поскольку возможности взаимодействия с внешними ресурсами еще более ограничены, по сравнению с теми, которые предоставляют среды Windows и Mac OS.

javax.swing.JMenu 1.2

- `JMenu(String label)`

  Создает меню

  Параметр `label` Метка меню в строке или в родительском меню

- `JMenuItem add(JMenuItem item)`

  Добавляет пункт (или целое меню).

Параметр `item` Добавляется пункт или отдельное меню

- `JMenuItem add(String label)`

  Добавляет пункт в меню.

  Параметр `label` Метка пункта меню

- `JMenuItem add(Action a)`

  Добавляет пункт и связанное с ним действие.

  Параметр `a` Объект `Action`, инкапсулирующий имя, пиктограмму *(по желанию) и обработчик (см. главу 8)*

- `void addSeparator()`

  Добавляет в меню разделитель.

- `JMenuItem insert(JMenuItem menu, int index)`

  Добавляет в указанную позицию меню новый пункт (или подменю).

  Параметры `menu` Меню для вставки

  `index` Позиция, в которую вставляется пункт

- `JMenuItem insert(Action a, int index)`

  Добавляет в указанную позицию меню новый пункт и связанный с ним объект `Action`.

  Параметры `a` Объект `Action`, инкапсулирующий имя, пиктограмму (по желанию) и обработчик

  `index` Позиция, в которую вставляется пункт

- `void insertSeparator(int index)`

  Добавляет в меню разделитель.

  Параметр `index` Позиция, в которой вставляется разделитель

- `void remove(int index)`

  Удаляет указанный пункт из меню.

  Параметр `index` Позиция, определяющая пункт, который подлежит удалению

- `void remove(JMenuItem item)`

  Удаляет указанный пункт меню.

  Параметр `item` Пункт меню, подлежащий удалению

`javax.swing.JMenuItem 1.2`

- `JMenuItem(String label)`

  Создает пункт меню с указанной меткой.

  Параметр `label` Метка пункта меню

- `JMenuItem(Action a)`

  Создает пункт меню и связывает с ним указанное действие.

Параметр a Объект Action, инкапсулирующий имя, пиктограмму (по желанию) и обработчик

javax.swing.AbstractButton 1.2

- `void setAction(Action a)` **1.3**

  Связывает с кнопкой или пунктом меню Объект Action.

  Параметр a Объект Action, инкапсулирующий имя, пиктограмму (по желанию) и обработчик

javax.swing.JFrame 1.2

- `void setJMenuBar(JMenuBar menubar)`

  Задает строку меню в указанном фрейме.

Пиктограммы в пунктах меню

Пункты меню очень похожи на кнопки. Фактически класс JMenuItem расширяет класс AbstractButton. Как и кнопки, меню могут иметь текстовую метку, пиктограмму или и то и другое одновременно. Пиктограмму можно указать в конструкторе JMenuItem(String, Icon) или задать с помощью метода setIcon(), унаследованного классом JMenuItem от класса AbstractButton.

```
JMenuItem cutItem = new JMenuItem("Cut",
   new ImageIcon("cut.gif"));
```

На рис. 9.22 показано меню с пиктограммами. По умолчанию названия пунктов меню располагаются справа от пиктограмм. Если вам хочется, чтобы справа находились пиктограммы, используйте метод setHorizontalTextPosition(), унаследованный классом JMenuItem от класса AbstractButton. Например, приведенное ниже выражение перемещает текст пункта меню влево от пиктограммы.

```
cutItem.setHorizontalTextPosition(SwingConstants.LEFT);
```

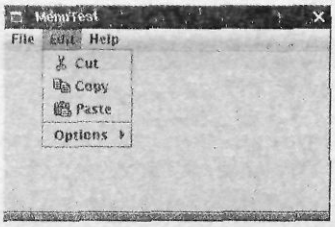


Рис. 9.22. Пиктограммы в пунктах меню

Пиктограмму можно связать с объектом Action.

```
cutAction.putValue(Action.SMALL_ICON, new ImageIcon("Cut.gif"));
```

Если пункт меню создается независимо от Action, то значением поля Action.NAME становится название пункта меню, а значением поля Action.SMALL_ICON — его пиктограмма.

Кроме того, пиктограмму можно задать в конструкторе класса AbstractAction.

```
cutAction = new
   AbstractAction("Cut", new ImageIcon("cut.gif"))
   {
      public void actionPerformed(ActionEvent event)
      {
         // Фрагмент, реализующий обработку.
      }
   };
```

javax.swing.JMenuItem 1.2

- `JMenuItem(String label, Icon icon)`

  Создает пункт меню с указанными меткой и пиктограммой.

javax.swing.AbstractButton 1.2

- `void setHorizontalTextPosition(int pos)`

  Задает взаимное расположение текста b пиктограммы.

  Параметр `pos` Константа `SwingConstants.RIGHT` (текст располагается справа от пиктограммы) или `SwingConstants.LEFT`

javax.swing.AbstractAction 1.2

- `AbstractAction(String name, Icon smallIcon)`

  Создает объект `AbstractAction` с указанными именем и пиктограммой.

Флажки опций и переключатели в меню

Пункты меню, представляющие собой *флажки опций* или *переключатели*, показаны на рис. 9.23. Когда пользователь щелкает на пункте меню, флажки автоматически устанавливаются или сбрасываются, а состояние переключателей изменяется в соответствии с выбранным пунктом.

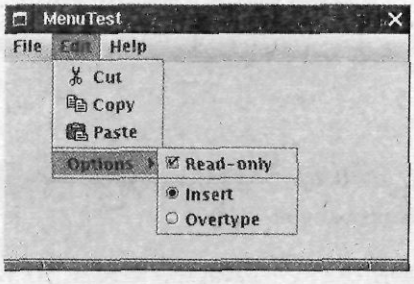


Рис. 9.23. Пункты меню с флажками опций и переключателями

Если не принимать во внимание внешний вид, флажки и опции можно рассматривать как обычные пункты меню. Рассмотрим, например, как создается пункт меню с флажком опции.

```
JCheckBoxMenuItem readonlyItem
   = new JCheckBoxMenuItem("Read-only");
optionsMenu.add(readonlyItem);
```

Пункты меню с переключателями работают точно так же, как и обычные переключатели. В меню нужно добавить группу переключателей. Когда выбирается один из них, все остальные автоматически отключаются.

```
ButtonGroup group = new ButtonGroup();
JRadioButtonMenuItem insertItem
   = JRadioButtonMenuItem("Insert");
insertItem.setSelected(true);
JRadioButtonMenuItem overtypeItem
   = new JRadioButtonMenuItem("Overtype");
group.add(insertItem);
group.add(overtypeItem);
optionsMenu.add(insertItem);
optionsMenu.add(overtypeItem);
```

В этих пунктах меню не обязательно точно определять, когда именно пользователь сделал выбор. Вместо этого для проверки текущего состояния пункта меню достаточно применить метод `isSelected()`. (Разумеется, это означает, что в какой-то переменной экземпляра придется хранить ссылку на данный пункт меню.) Кроме того, задать состояние пункта меню можно с помощью метода `setSelected()`.

javax.swing.JCheckBoxMenuItem 1.2

- `JCheckBoxMenuItem(String label)`

  Создает пункт меню, представляющий собой флажок опции с заданной меткой.

- `JCheckBoxMenuItem(String label, boolean state)`

  Создает пункт меню, представляющий собой флажок опции с заданными меткой и состоянием (если пункт выбран, значение переменной `state` равно `true`).

javax.swing.JRadioButtonMenuItem 1.2

- `JRadioButtonMenuItem(String label)`

  Создает пункт меню, представляющий собой переключатель с заданной меткой.

- `JRadioButtonMenuItem(String label, boolean state)`

  Создает пункт меню, представляющий собой переключатель с заданными меткой и состоянием (если пункт выбран, значение переменной `state` равно `true`).

javax.swing.AbstractButton 1.2

- `boolean isSelected()`

  Возвращает состояние пункта (значение `true`, если пункт выбран).

- `void setSelected(bboolean state)`

  Задает состояние пункта.

Контекстные меню

Всплывающие, или *контекстные, меню* (pop-up menu) — это меню, не связанные со строкой, а отображающиеся в произвольно выбранной позиции на экране (рис. 9.24).

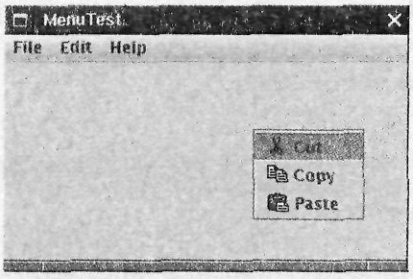


Рис. 9.24. Всплывающее меню

Всплывающее меню создается точно так же, как и обычное, за одним исключением — оно не имеет заголовка.

```
JpopupMenu popup = new JPopupMenu();
```

Пункты меню добавляются, как обычно.

```
JmenuItem item = new JmenuItem("Cut");
item.addActionListener(listener);
popup.add(item);
```

В отличие от строки меню, которая всегда находится в верхней части фрейма, всплывающее меню нужно принудительно выводить на экран с помощью метода `show()`. При этом задается родительский компонент и расположение всплывающего меню в его системе координат.

```
popup.show(panel, x, y);
```

Обычно всплывающее меню отображается на экране, когда пользователь щелкает специально предназначенной для этого кнопкой. В системах Windows и Linux это правая кнопка мыши. Для вызова меню после щелчка кнопкой вызывается следующий метод:

```
component.setComponentPopupMenu(popup);
```

Часто приходится размещать компонент внутри другого компонента, с которым связано всплывающее меню. Для того чтобы дочерний компонент наследовал меню родительского компонента, надо вызвать метод

```
child.setInheritsPopupMenu(true);
```

Приведенные выше методы были реализованы в JDK 5.0 для того, чтобы исключить зависимость программиста от особенностей работы с контекстным меню в конкретной системе. В предшествующих версиях JDK приходилось устанавливать обработчик событий, связанных с мышью, и задавать представленный ниже код как в методе `mousePressed()`, так и в методе `mouseReleased()`.

```
if (popup.isPopupTrigger(event))
   popup.show(event.getComponent(), event.getX(), event.getY());
```

Причина в том, что в некоторых системах всплывающее меню отображается при нажатии кнопки мыши, а в других — при отпускании кнопки.

javax.swing.JPopupMenu 1.2

- `void show(Component c, int x, int y)`

  Отображает всплывающее меню.

  Параметры `c` Компонент, соответствующий всплывающему меню

  `x, y` Координаты (в системе компонента `c`) левого верхнего угла всплывающего меню

- `boolean isPopupTrigger(MouseEvent event)` **1.3**

  Возвращает значение `true`, если событие, связанное с мышью, соответствует вызову всплывающего меню.

java.awt.event.MouseEvent 1.1

- `boolean isPopupTrigger()`

  Возвращает значение `true`, если событие, связанное с мышью, соответствует вызову всплывающего меню.

javax.swing.JComponent 1.2

- `void setComponentPopupMenu(JPopupMenu popup)` **5.0**
- `JPopup getComponentPopupMenu()` **5.0**

  Устанавливает или возвращает всплывающее меню, связанное с компонентом.

- `void setInheritsPopupMenu(boolean b)` **5.0**
- `boolean getInheritsPopupMenu()` **5.0**

  Устанавливает или возвращает свойство *inheritsPopupMenu*. Если данное свойство установлено, а вместо всплывающего меню компоненту соответствует значение `null`, используется меню родительского компонента.

Клавиши быстрого доступа и клавишные сокращения

Опытному пользователю удобно выбирать пункты меню с помощью *клавиш быстрого доступа* (keyboar mnemonics). Связать пункт меню с клавишей быстрого доступа можно, задав эту клавишу в конструкторе пункта.

```
JMenuItem cutItem = new JmenuItem("Cut", "T");
```

Буква в названии пункта меню, соответствующая клавише быстрого доступа, выделяется подчеркиванием (рис. 9.25). Например, если клавиша быстрого доступа задана с помощью приведенного выше выражения, то в пункте Cut буква t будет подчеркнута. Теперь для выбора этого пункта меню пользователю достаточно нажать клавишу <T>. (Если буква, соответствующая назначенной клавише, не входит в название пункта меню, она не отображается на экране, но при ее нажатии этот пункт все равно будет выбран. Естественно, польза от таких "невидимых" клавиш сомнительна.)

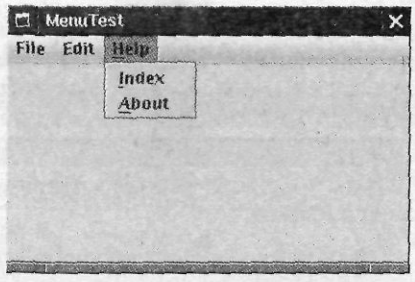


Рис. 9.25. Так выглядят пункты меню, для которых назначены клавиши быстрого доступа

Задавая клавишу быстрого доступа, не всегда целесообразно выделять первое вхождение соответствующей буквы в название пункта меню. Так, например, если для пункта Save As назначена клавиша <A>, то гораздо уместнее выделить подчеркиванием букву A в слове As (Save As). В JDK 1.4 и более поздних версиях можно указывать символ, который должен быть выделен, с помощью метода `setDisplayedMnemonicIndex()`.

Имея объект класса `Action`, клавишу быстрого доступа можно назначить с помощью значения поля `Action.MNEMONIC_KEY`.

```
cutAction.putValue(Action.MNEMONIC_KEY, new Integer('T'));
```

Букву, соответствующую клавише быстрого доступа, следует задавать только в конструкторе пункта меню (но не в конструкторе меню в целом). Чтобы связать какую-либо клавишу с меню в целом, нужно вызвать метод `setMnemonic()`.

```
JMenu helpMenu = new JMenu("Help");
helpMenu.setMnemonic('H');
```

Теперь, чтобы выбрать меню в строке, нужно нажать клавишу <Alt> вместе с назначенной буквой. Например, чтобы выбрать меню Help (см. рис. 9.25), нужно нажать клавиши <Alt+H>.

Клавиши быстрого доступа позволяют выбрать пункт в уже открытом меню. В противоположность этому, *клавишные сокращения* (accelerator) позволяют выбрать пункт, не открывая меню. Например, во многих программах предусмотрены сочетания клавиш <Ctrl+O> и <Ctrl+S> для пунктов Open и Save меню File. Для связывания сочетания клавиш с пунктом меню используется метод `setAccelerator()`. В качестве параметра он получает объект `Keystroke`. Например, приведенный ниже вызов назначает клавишное сокращение <Ctrl+O> для пункта меню `openItem`.

```
openItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_O,
   InputEvent.CTRL_MASK));
```

Когда пользователь нажимает требуемое сочетание клавиш, он автоматически выбирает соответствующий пункт меню и генерирует событие, так же, как если бы он выбирал пункт меню обычным способом. Клавишные сокращения можно связывать только с пунктами меню, но не с меню в целом. Они не открывают меню, а лишь генерируют событие, связанное с указанным пунктом.

В принципе, клавишные сокращения для пунктов меню добавляются так же, как и для остальных компонентов пакета Swing. Однако, если клавишное сокращение назначено, соответствующие клавиши автоматически отображаются в меню (рис. 9.26).

В операционной системе Windows сочетание клавиш <Alt+F4> закрывает окно. Однако сочетание не имеет отношение к языку Java. Это клавишное сокращение определено в операционной системе и всегда генерирует событие `WindowClosing` для активного окнв, независимо от того, есть ли в меню пункт Close.

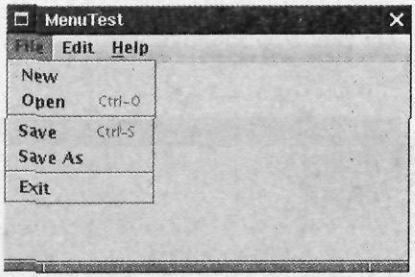


Рис. 9.26. Клавишные сокращения

`javax.swing.JMenuItem 1.2`

- `JMenuItem(String label, int mnemonic)`

  Создает пункт меню с указанной меткой и клавишей быстрого доступа.

  Параметры `label` Метка пункта меню

  `mnemonic` Мнемоническое обозначение пункта, соответствующее клавише быстрого доступа; в метке этот символ подчеркивается

- `void setAccelerator(Keystroke k)`

  Задает клавишное сокращение `k` для указанного пункта меню. Соответствующее сочетание клавиш отображается в меню рядом с меткой.

`javax.swing.AbstractButton 1.2`

- `void setMnemonic(char mnemonic)`

  Назначает клавишу быстрого доступа для кнопки. В метке кнопки этот символ подчеркивается.

  Параметр `mnemonic` Символ, соответствующий клавише быстрого доступа, назначенный для кнопки

- `void setDisplayedMnemonicIndex(int index) 1.4`

  Задает расположение символа, который должен быть выделен подчеркиванием. Данный метод вызывается в том случае, если выделять первое вхождение символа, соответствующего клавише быстрого доступа, нецелесообразно.

  Параметр `index` Индекс символа, который должен быть выделен

Блокирование и разблокирование пунктов меню

Бывают случаи, когда некоторые пункты меню должны выбираться лишь в определенных ситуациях. Например, если документ открыт лишь для чтения, то пункт меню Save не имеет смысла. Разумеется, этот пункт можно удалить с помощью метода `JMenu.remove()`, но пользователя может удивить постоянно изменяющееся меню. Лучше все же заблокировать пункты меню, временно лишив пользователя возможности выполнять ряд операций. На рис. 9.27 заблокированный пункт меню выделен светло-серым цветом.

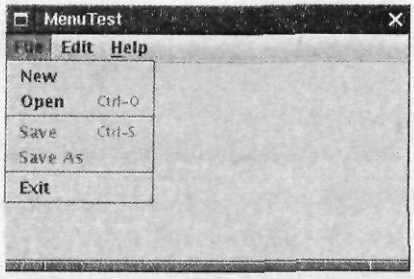


Рис. 9.27. Заблокированные пункты меню

Для того чтобы разблокировать или заблокировать пункт меню, используют метод `setEnabled()`.

```
saveItem.setEnabled(false);
```

Существуют две стратегии блокирования и разблокирования пунктов меню. При каждом изменении состояния программы можно вызывать метод `setEnabled()`, применяя его к соответствующему пункту меню. Например, открыв документ только для чтения, можно заблокировать пункты Save и Save As. Однако, пользуясь этой стратегией, программист рискует "засорить" код фрагментами, предназначенными для изменения состояния пунктов меню. Гораздо лучше совсем не беспокоиться о состоянии пунктов меню в основной части программы и *задавать их параметры непосредственно перед отображением на экране.* Для этого нужно зарегистрировать обработчик событий, связанных с выбором меню. В пакете `javax.swing.event` содержится интерфейс `MenuListener`, в котором объявлены три метода.

```
void menuSelected(MenuEvent event)
void menuDeselected(MenuEvent event)
void menuCanceled(MenuEvent event)
```

Метод `menuSelected()` вызывается *перед* отображением меню на экране. Это самый подходящий момент для того, чтобы заблокировать или разблокировать пункты меню. В приведенном ниже фрагменте кода показано, как блокируются и разблокируются пункты Save и Save As, в зависимости от состояния флажка Read Only.

```
public void menuSelected(MenuEvent event)
{
   saveAction.setEnabled(!readonlyItem.isSelected());
   saveAsAction.setEnabled(!readonlyItem.isSelected());
}
```

Блокировка пунктов меню непосредственно перед его отображением на экране — отличная идея, однако она не работает для пунктов меню, связанных с клавишными сокращениями. При нажатии соответствующего сочетания клавиш меню не открывается, поэтому пункт меню не будет заблокирован.

javax.s

- `void setEnabled(boolean b)`

  Блокирует и разблокирует пункт меню.

javax.swing.event.MenuListener 1.2

- `void menuSelected(menuEvent e)`

  Вызывается, когда меню уже выбрано, но еще не отображается на экране.

- `void menuDeselected(menuEvent e)`

  Вызывается, когда меню уже закрыто.

- `void menuCanceled(menuEvent e)`

  Вызывается, когда действия с меню отменены, например, если пользователь щелкнул мышью за пределами меню.

В листинге 9.11 приведен пример программы, генерирующей набор меню. Продемонстрированы все особенности меню, описанные в этом разделе: вложенные меню, заблокированные пункты, флажки опций и переключатели в пунктах меню, а также клавиши быстрого доступа и клавишные сокращения.

Листинг 9.11. Содержимое файла `MenuTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class MenuTest
{
   public static void main(String[] args)
   {
      MenuFrame frame = new MenuFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм со строкой меню.
*/
class MenuFrame extends JFrame
{
   public MenuFrame()
   {
```

```
setTitle("MenuTest");
setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

JMenu fileMenu = new JMenu("File");
JMenuItem newItem = fileMenu.add(new TestAction("New"));

// Клавишные сокращения.

JMenuItem openItem = fileMenu.add(new TestAction("Open"));
openItem.setAccelerator(KeyStroke.getKeyStroke(
   KeyEvent.VK_O, InputEvent.CTRL_MASK));

fileMenu.addSeparator();

saveAction = new TestAction("Save");
JMenuItem saveItem = fileMenu.add(saveAction);
saveItem.setAccelerator(KeyStroke.getKeyStroke(
   KeyEvent.VK_S, InputEvent.CTRL_MASK));

saveAsAction = new TestAction("Save As");
JMenuItem saveAsItem = fileMenu.add(saveAsAction);
fileMenu.addSeparator();

fileMenu.add(new
   AbstractAction("Exit")
   {
      public void actionPerformed(ActionEvent event)
      {
         System.exit(0);
      }
   });

// Флажки опций и переключатели в составе меню.

readonlyItem = new JCheckBoxMenuItem("Read-only");
readonlyItem.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         boolean saveOk = !readonlyItem.isSelected();
         saveAction.setEnabled(saveOk);
         saveAsAction.setEnabled(saveOk);
      }
   });

ButtonGroup group = new ButtonGroup();

JRadioButtonMenuItem insertItem =
   new JRadioButtonMenuItem("Insert");
insertItem.setSelected(true);
JRadioButtonMenuItem overtypeItem =
   new JRadioButtonMenuItem("Overtype");

group.add(insertItem);
group.add(overtypeItem);
```

```
// Пиктограммы в составе меню.

Action cutAction = new TestAction("Cut");
cutAction.putValue(Action.SMALL_ICON,
   new ImageIcon("cut.gif"));
Action copyAction = new TestAction("Copy");
copyAction.putValue(Action.SMALL_ICON,
    new ImageIcon("copy.gif"));
Action pasteAction = new TestAction("Paste");
pasteAction.putValue(Action.SMALL_ICON,
   new ImageIcon("paste.gif"));

JMenu editMenu = new JMenu("Edit");
editMenu.add(cutAction);
editMenu.add(copyAction);
editMenu.add(pasteAction);

// Вложенные меню.

JMenu optionMenu = new JMenu("Options");

optionMenu.add(readonlyItem);
optionMenu.addSeparator();
optionMenu.add(insertItem);
optionMenu.add(overtypeItem);

editMenu.addSeparator();
editMenu.add(optionMenu);

// Клавиши быстрого доступа.

JMenu helpMenu = new JMenu("Help");
helpMenu.setMnemonic('H');

JMenuItem indexItem = new JMenuItem("Index");
indexItem.setMnemonic('I');
helpMenu.add(indexItem);

// Задать клавишу быстрого доступа можно также
// с помощью объекта Action.
Action aboutAction = new TestAction("About");
aboutAction.putValue(Action.MNEMONIC_KEY, new Integer('A'));
helpMenu.add(aboutAction);

// Включение меню верхнего уровня в строку меню.

JMenuBar menuBar = new JMenuBar();
setJMenuBar(menuBar);

menuBar.add(fileMenu);
menuBar.add(editMenu);
menuBar.add(helpMenu);

// Всплывающее меню
```

```
      popup = new JPopupMenu();
      popup.add(cutAction);
      popup.add(copyAction);
      popup.add(pasteAction);

      JPanel panel = new JPanel();
      panel.setComponentPopupMenu(popup);
      add(panel);

      // Следующая строка необходима из-за ошибки 4966109
      panel.addMouseListener(new MouseAdapter() {});
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   private Action saveAction;
   private Action saveAsAction;
   private JCheckBoxMenuItem readonlyItem;
   private JPopupMenu popup;
}

/**
   Обработчик события, который выводит в System.out
   имя Action.
*/
class TestAction extends AbstractAction
{
   public TestAction(String name) { super(name); }

   public void actionPerformed(ActionEvent event)
   {
      System.out.println(getValue(Action.NAME) + " selected.");
   }
}
```

Панели инструментов

Панель инструментов представляет собой набор кнопок, обеспечивающих быстрый доступ к наиболее часто используемым командам (рис. 9.28).

Что отличает панель инструментов от остальных элементов пользовательского интерфейса? Ее можно перетаскивать на любую из четырех сторон фрейма (рис. 9.29). При отпускании кнопки мыши панель инструментов зафиксируется на новом месте (рис. 9.30).

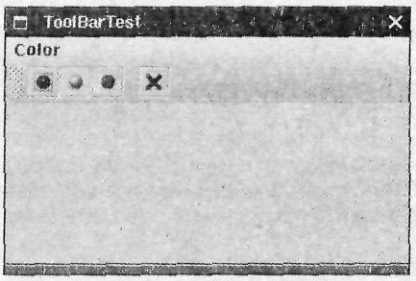


Рис. 9.28. Панель инструментов

Перетаскивание панели инструментов допускается, только если она находится в контейнере, управляемом диспетчером компоновки `BorderLayout` рамок или любым другим диспетчером, поддерживающим области `North`, `East`, `South` и `West`.

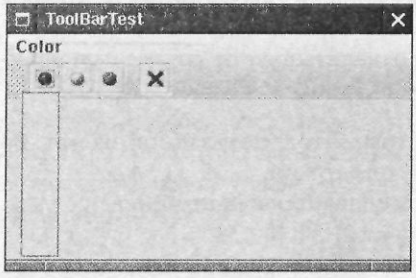


Рис. 9.29. Процесс перетаскивания панели инструментов

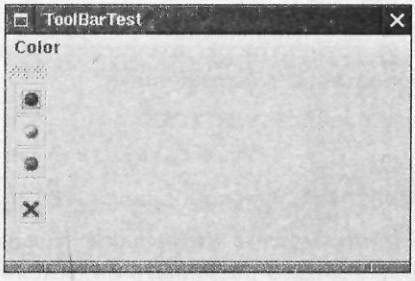


Рис. 9.30. В результате перетаскивания панель инструментов размещена вдоль левой стороны фрейма

Панель инструментов может быть даже независимой от фрейма. Такая панель содержится в своем собственном фрейме (рис. 9.31). Если пользователь закрывает фрейм, содержащий независимую панель инструментов, она перемещается в исходный фрейм.

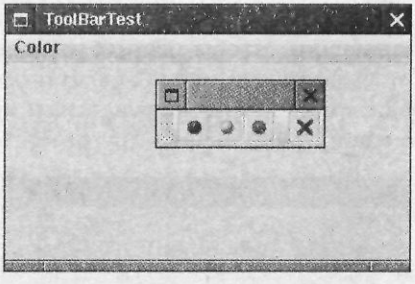


Рис. 9.31. Независимая панель инструментов

Панель инструментов легко программировать. Ниже приведен пример создания панели и добавления к ней компонента.

```
JToolBar bar = new JToolBar();
bar.add(blueButton);
```

В классе JToolBar также есть методы, предназначенные для добавления объекта класса Action.

```
bar.add(blueAction);
```

Пиктограмма, соответствующая такому объекту, отображается на панели инструментов.

Группы кнопок можно отделять друг от друга с помощью разделителей.

```
bar.addSeparator();
```

Например, на панели инструментов, показанной на рис. 9.28, есть разделитель, отделяющий третью кнопку от четвертой.

Панель инструментов помещается в контейнер.

```
add(bar, BorderLayout.NORTH);
```

Можно также указать заголовок панели, который появится, когда панель будет отображаться независимо от фрейма.

```
bar = new JToolBar(titleString);
```

По умолчанию панель инструментов располагается горизонтально. Чтобы сделать ее вертикальной, нужно использовать выражение

```
bar = new JToolBar(SwingCostants.VERTICAL)
```

или

```
bar = new JToolBar(titleString, SwingCostants.VERTICAL)
```

Чаще всего на панель инструментов помещают кнопки. Однако ограничений на вид компонентов, которые можно размещать на панели инструментов, не существует. Например, на нее можно поместить раскрывающийся список.

Примеры инструментальных панелей можно найти на Web-странице `http://developer.java.sun.com/developer/techDocs/hi/repository`.

Подсказки

Панели инструментов имеют существенный недостаток. По внешнему виду кнопки трудно догадаться о ее назначении. Чтобы решить эту проблему, были созданы *подсказки* (tool tips), которые появляются на экране, когда курсор задерживается на кнопке. Текст подсказки отображается внутри закрашенного прямоугольника (рис. 9.32). Когда пользователь перемещает курсор мыши в другое место, подсказка исчезает.

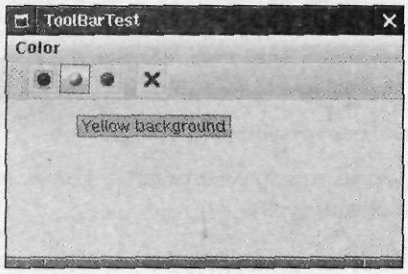


Рис. 9.32. Подсказка

В пакете Swing подсказки можно добавлять к любому объекту класса JComponent, вызывая метод setToolTipText().

```
exitButton.setToolTipText("Exit");
```

Если вы используете объекты класса Action, подсказка связывается с ключом SHORT_DESCRIPTION.

```
exitAction.putValue(Action.SHORT_DESCRIPTION, "Exit");
```

В листинге 9.12 приведена программа, показывающая, как одни и те же объекты класса Action можно добавлять в меню и на панель инструментов. Обратите внимание на то, что имена появляются на экране и в виде названий пунктов меню, и в виде подсказок к элементам панели инструментов.

Листинг 9.12. Содержимое файла ToolBarTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import javax.swing.*;

public class ToolBarTest
{
   public static void main(String[] args)
   {
      ToolBarFrame frame = new ToolBarFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с панелью инструментов и меню для изменения цвета.
*/
class ToolBarFrame extends JFrame
{
   public ToolBarFrame()
   {
      setTitle("ToolBarTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление панели с изменяемым цветом.

      panel = new JPanel();

      // Создание объектов Action

      Action blueAction = new ColorAction("Blue",
         new ImageIcon("blue-ball.gif"), Color.BLUE);
      Action yellowAction = new ColorAction("Yellow",
         new ImageIcon("yellow-ball.gif"), Color.YELLOW);
      Action redAction = new ColorAction("Red",
         new ImageIcon("red-ball.gif"), Color.RED);

      Action exitAction = new
```

```
            AbstractAction("Exit", new ImageIcon("exit.gif"))
            {
               public void actionPerformed(ActionEvent event)
               {
                  System.exit(0);
               }
            };
         exitAction.putValue(Action.SHORT_DESCRIPTION, "Exit");

         // Заполнение панели инструментов.

         JToolBar bar = new JToolBar();
         bar.add(blueAction);
         bar.add(yellowAction);
         bar.add(redAction);
         bar.addSeparator();
         bar.add(exitAction);
         add(bar, BorderLayout.NORTH);

         // Заполнение меню.

         JMenu menu = new JMenu("Color");
         menu.add(yellowAction);
         menu.add(blueAction);
         menu.add(redAction);
         menu.add(exitAction);
         JMenuBar menuBar = new JMenuBar();
         menuBar.add(menu);
         setJMenuBar(menuBar);
      }

      public static final int DEFAULT_WIDTH = 300;
      public static final int DEFAULT_HEIGHT = 200;

      private JPanel panel;

      /**
         ColorAction устанавливает требуемый цвет фона.
      */
      class ColorAction extends AbstractAction
      {
         public ColorAction(String name, Icon icon, Color c)
         {
            putValue(Action.NAME, name);
            putValue(Action.SMALL_ICON, icon);
            putValue(Action.SHORT_DESCRIPTION, name + " background");
            putValue("Color", c);
         }

         public void actionPerformed(ActionEvent event)
         {
            Color c = (Color) getValue("Color");
            panel.setBackground(c);
         }
      }
}
```

javax.swing.JToolBar 1.2

- `JToolBar()`
- `JToolBar(String titleString)`
- `JToolBar(int orientation)`
- `JToolBar(String titleString, int orientation)`

  Создает панель инструментов, имеющую заданную строку заголовка и ориентацию. Параметр `orientation` может принимать значения `SwingConstants.HORIZONTAL` (по умолчанию) и `SwingConstants.VERTICAL`.

- `JButton add(Action a)`

  Создает новую кнопку на панели инструментов с именем, пиктограммой и обратным вызовом. Кнопка добавляется в конец панели инструментов.

- `void addSeparator()`

  Добавляет разделитель в конец панели инструментов.

javax.swing.JComponent 1.2

- `void setToolTipText(String text)`

  Задает текст, подлежащий выводу на экран в качестве подсказки, когда курсор мыши останавливается на компоненте.

Расширенные средства компоновки

До сих пор при разработке интерфейса для наших приложений мы пользовались лишь диспетчерами компоновки `BorderLayout`, `FlowLayout` и `GridLayout`. Для решения более сложных задач этого недостаточно. В этом разделе описываются все диспетчеры компоновки, предусмотренные стандартной библиотекой языка Java.

У программистов, работающих на платформе Windows, может возникнуть вопрос: отчего в языке Java столько внимания уделяется диспетчерам компоновки. Ведь во многих интегрированных средах программирования достаточно перетащить компоненты в диалоговое окно и выровнять их, используя средства редактирования. При разработке большого проекта программист может вообще не думать о компоновке — квалифицированный дизайнер возьмет на себя все заботы об интерфейсе.

Такой подход имеет существенный недостаток. Если размеры элементов изменяются, приходится перекомпоновывать их вручную. Отчего компоненты могут изменить свои размеры? Это происходит в двух случаях. Во-первых, пользователь может выбрать более крупный шрифт для надписей на кнопках и других элементов диалогового окна. Если вы попробуете это сделать, работая под управлением операционной системы Windows, то убедитесь сами, что многие приложения крайне неудачно справляются с этой задачей. Кнопки не увеличиваются, и более крупный шрифт "втискивается" в те же самые размеры. Такая же проблема может возникнуть при переводе интерфейса на иностранный язык. Например, слово Cancel на немецкий язык переводится как Abbrechen. Если кнопка разработана так, что в ней помещается слово Cancel, то немецкий перевод на той же кнопке будет усечен.

Почему кнопки, созданные на платформе Wndows, не увеличиваются, чтобы поместить метки на своей поверхности? Это происходит потому, что разработчики интерфейса не предусмотрели никаких команд, задающих направление, в котором

должны увеличиваться размеры кнопок. После того как элементы интерфейса будут перемещены в окно и выровнены, редактор диалога просто забывает координаты и размеры всех компонентов. Таким образом, информация, позволяющая судить о том, *почему* компоненты были расположены именно таким образом, теряется.

Диспетчеры компоновки в языке Java справляются с размещением компонентов намного лучше. По сути, компоновка сводится к созданию инструкций, описывающих *отношения между компонентами. Это особенно важно при работе с библиотекой* AWT, использующей платформенно-зависимые элементы пользовательского интерфейса. Размер кнопки или списка зависит от платформы, а разработчики приложения *или аплета не могут заранее знать, на какой платформе будет отображаться их пользо*вательский интерфейс. Если стиль интерфейса, например Metal, заранее задан, то внешний вид приложения будет абсолютно одинаков на всех платформах. Однако, если программист хочет дать пользователю возможность самому задавать стиль интерфейса, придется положиться на гибкость диспетчеров компоновки.

Разумеется, для создания сложных интерфейсов нужны более мощные средства управления по сравнению с диспетчерами `BorderLayout`, `FlowLayout` и `GridLayout`. В этом разделе мы обсудим средства компоновки, которые предусмотрены в стандартной библиотеке языка Java. Использование нескольких панелей, связанных с различными диспетчерами компоновки, позволяет программисту полностью контролировать внешний вид окна приложения.

Если вы не нашли такой диспетчер компоновки, который полностью соответствовал бы вашим потребностям, разбейте окно на несколько панелей и скомпонуйте компоненты на каждой из них. Затем объедините панели с помощью диспетчера компоновки.

Рассмотрим основные принципы компоновки. Как известно, в библиотеке AWT *компоненты* располагаются в *контейнерах*. Кнопки, поля редактирования и другие элементы пользовательского интерфейса являются компонентами и расширяют класс `Component`. Контейнеры, например, панели, сами могут располагаться внутри другого контейнера, так как класс `Container` является подклассом класса `Component`. На рис. 9.33 показана иерархия наследования для класса `Component`.

Обратите внимание на то, что некоторые объекты принадлежат классам, расширяющим класс `Component`, хотя они не являются элементами пользовательского интерфейса и не могут быть вставлены в контейнеры. В частности, окна верхнего уровня, такие как `JFrame` и `JApplet`, не могут находиться в другом окне или на другой панели.

Как вы уже знаете, чтобы поместить компоненты в контейнер, сначала нужно указать диспетчер компоновки. Например, приведенное ниже выражение связывает с панелью диспетчер `GridLayout`.

```
panel.setLayout(new GridLayout(4, 4));
```

Итак, диспетчер компоновки задан. Теперь нужно добавить компоненты в контейнер. Метод `add()` передает диспетчеру компоновки компонент и, возможно, инструкции по его размещению.

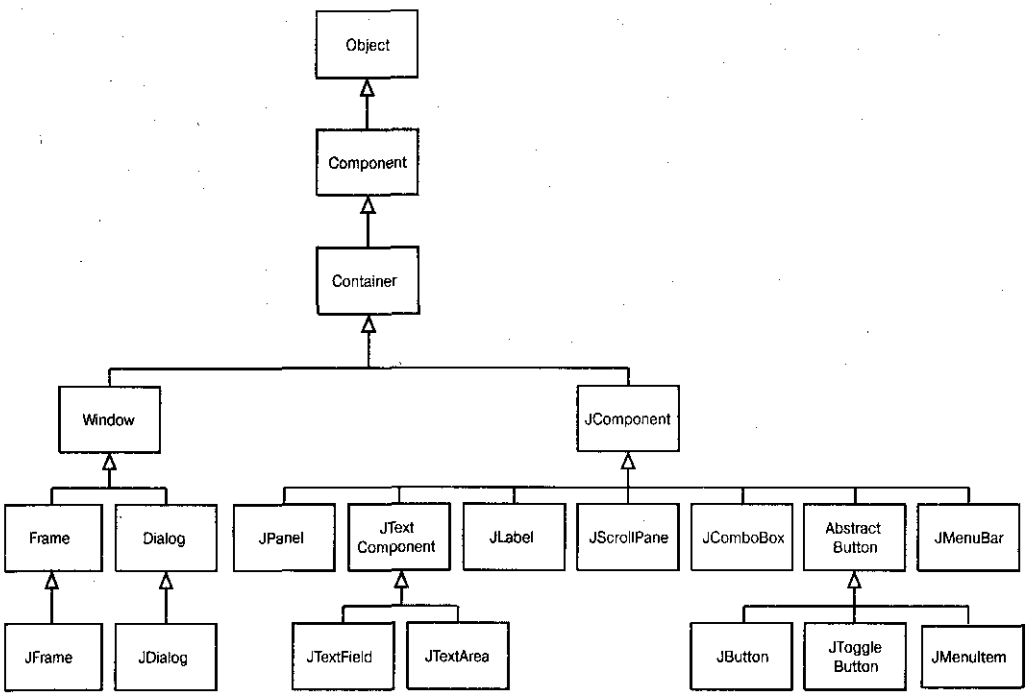


Рис. 9.33. Иерархия наследования для класса Component

Если вы используете диспетчер компоновки `BorderLayout`, ему передается строка, определяющая область для размещения компонента.

```
panel.add(new JTextField(), BorderLayout.SOUTH);
```

При работе с диспетчером `GridLayout` компоненты нужно включать последовательно один за другим.

```
panel.add(new JCheckBox("italic"));
panel.add(new JCheckBox("bold"));
```

При этом они будут располагаться в порядке, напоминающем ячейки электронной таблицы. Однако размеры всех строк и столбцов одинаковы, что не всегда удобно в практической работе.

Для того чтобы преодолеть это ограничение, в AWT предусмотрен диспетчер `GridBagLayout`. Он также позволяет расставлять компоненты по строкам и столбцам, но при этом один компонент может занимать несколько строк и столбцов. Этот диспетчер компоновки очень гибок, но и сложен. Далее мы покажем, как можно справиться с проблемами, возникающими при его использовании.

Предпринимая безуспешную попытку создать диспетчер компоновки, который избавил бы программиста от необходимости разбираться с особенностями `GridBagLayout`, разработчики библиотеки Swing создали диспетчер `BoxLayout`, который расставляет последовательность компонентов в горизонтальном или вертикальном направлении. Горизонтальная расстановка компонентов напоминает действия диспетчера `FlowLayout`;

однако компоненты не переносятся на новую строку, если предыдущая оказалась заполненной. Включая несколько горизонтальных блоков внутри вертикального блока, программист может задать определенный порядок размещения компонентов в двухмерной области. Однако, поскольку каждый блок компонуется независимо, не удается выравнивать компоненты одновременно по вертикали и по горизонтали.

В библиотеке Swing также реализован диспетчер `OverlayLayout`, позволяющий размещать компоненты один поверх другого. Обычно такой вариант компоновки не приносит много пользы, и мы его обсуждать не будем.

В заключение диспетчер `CardLayout`, который размещает один контейнер поверх другого. Поскольку в библиотеке Swing для этой цели имеется контейнер с вкладками `JTabbedPane` (см. том 2), такой вид компоновки мы также не рассматриваем.

Мы закончим наше обсуждение, показав, как можно обойтись без диспетчеров компоновки вообще и разместить компоненты вручную, а также создать свой собственный диспетчер компоновки.

Диспетчер компоновки BoxLayout

Диспетчер `BoxLayout` размещает компоненты в строках и столбцах, причем обеспечивает большую степень гибкости по сравнению с диспетчером `GridLayout`. Существует даже контейнер (класс `Box`) с которым по умолчанию связывается диспетчер компоновки `BoxLayout` (как вы знаете, класс `JPanel` использует диспетчер `FlowLayout`). Разумеется, для класса `JPanel` можно назначить любой другой диспетчер, однако проще всего непосредственно работать с контейнером `Box`. Класс `Box` содержит также большое количество полезных статических методов.

Чтобы создать новый контейнер, нужно вызвать один из следующих методов:

```
Box b = Box.createHorizontalBox();
Box b = Box.createVerticalBox();
```

Затем компоненты добавляются к контейнеру, как обычно.

```
b.add(okButton);
b.add(cancelButton);
```

В горизонтальном блоке компоненты располагаются слева направо, а в вертикальном — сверху вниз. Рассмотрим подробнее горизонтальную компоновку.

С каждым компонентом связываются три размера.

- *Рекомендуемый размер* (preferred size) — ширина и высота.
- *Максимальный размер* (maximum size) — наибольшие ширина и высота.
- *Минимальный размер* (minimum size) — наименьшие ширина и высота.

В процессе работы диспетчер `BoxLayout` выполняет следующие действия.

1. Вычисляет максимальную (!) высоту самого высокого компонента.
2. Делает попытку увеличить высоту всех компонентов до максимальной высоты самого высокого компонента.
3. Если какой-либо компонент не увеличился до требуемой высоты, то с помощью метода `getAlignmentY()` запрашивается значение *y-выравнивания*. Эта величина представляет собой число с плавающей точкой в диапазоне от 0 (выравнивание по верхней границе) до 1 (выравнивание по нижней границе).

По умолчанию в классе Component это число равно 0,5 (выравнивание по центру). Это значение используется для вертикального выравнивания компонента.

4. Определяет рекомендуемую ширину каждого компонента. Все эти величины складываются, образовывая общую рекомендуемую ширину.
5. Если общая рекомендуемая ширина не превышает ширину блока, то размеры компонентов увеличиваются, но так, чтобы максимально возможные значения ширины не превышались. Затем компоненты размещаются слева направо, причем расстояние между ними не увеличивается. Если общая рекомендуемая ширина превышает ширину блока, размеры компонентов уменьшаются так, чтобы ширина не стала меньше минимально возможной. Если не все компоненты умещаются в своих минимальных размерах, то некоторые из них не будут отображаться на экране.

Аналогично выполняется компоновка вертикального блока.

Тот факт, что диспетчер BoxLayout пытается увеличить размеры компонентов сверх рекомендуемых, можно рассматривать как недостаток. В частности, максимальные длина и высота текстовых полей ограничиваются константой Integer.MAX_VALUE. Таким образом, они могут увеличиваться практически без ограничений. Если поместить текстовое поле в блок, его размеры могут стать гигантскими. Чтобы избежать этого, устанавливайте максимальный размер равным рекомендуемому, вызвав следующий метод:

```
textField.setMaximumSize(textField.getPreferredSize());
```

Заполнители

По умолчанию компоненты в блоке располагаются вплотную друг к другу. (В отличие от диспетчера FlowLayout, здесь нет понятия "расстояние между компонентами".) Для того чтобы раздвинуть компоненты, добавляются невидимые *заполнители* (filler). Есть три вида заполнителей:

- распорки (strut);
- фиксированные области (rigid area);
- склейки (glue).

Распорка добавляет определенное пространство между компонентами. Ниже показано, как можно добавить десять пикселей между двумя компонентами в горизонтальном блоке.

```
b.add(label);
b.add(Box.createHorizontalStrut(10));
b.add(textField);
```

Для того чтобы увеличить расстояние между компонентами в горизонтальных или вертикальных блоках, нужно вставить распорки. В горизонтальный блок можно вставить вертикальную распорку, но на горизонтальную компоновку это не повлияет, а приведет лишь к установке минимальной высоты блока.

Фиксированная область похожа на пару распорок. Она также раздвигает смежные компоненты, но, кроме того, определяет минимальное значение высоты или ширины в направлении, перпендикулярном направлению следования элементов. Например,

следующее выражение добавляет невидимую фиксированную область с заданными минимальной, рекомендуемой и максимальной шириной, равными 5 пикселям, и высотой, равной 20 пикселям, а также задает выравнивание по центру.

```
b.add(Box.createRigidArea(new Dimension(5, 20));
```

Если добавить эту область в горизонтальный блок, она сработает как распорка, имеющая ширину 5 пикселей, а также задаст минимальную высоту блока равной 20 пикселям.

Добавляя распорки, вы раздвигаете смежные компоненты на фиксированное расстояние. Добавление склеек приводит к тому, что компоненты раздвигаются на *максимально возможное расстояние*. Невидимая склейка расширяет блоки, отодвигая компоненты друг от друга, чтобы они заполнили все доступное пространство. (Совершенно непонятно, почему разработчики назвали такой заполнитель "склейкой" – более подходящим названием было бы слово "пружина".)

Вот как, например, можно заполнить блок двумя кнопками с помощью склейки:

```
b.add(button1);
b.add(Box.createGlue());
b.add(button2);
```

Если блок не содержит других компонентов, кнопка `button1` будет располагаться как можно дальше влево, а кнопка `button2` — вправо.

Программа, приведенная в листинге 9.13, размещает метки, поля редактирования и кнопки, используя вертикальную и горизонтальную компоновки. Каждый ряд компонентов помещается в горизонтальный блок. Метки отделяются от полей редактирования распорками. Склейки отталкивают две кнопки друг от друга. Три горизонтальных блока помещаются в вертикальный блок, в котором склейка отталкивает кнопку к нижней границе блока (рис. 9.34).

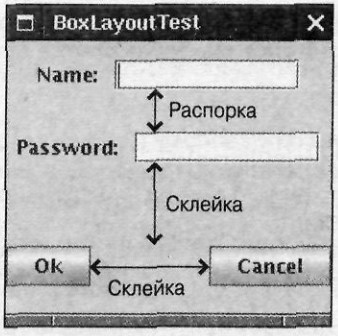


Рис. 9.34. Компоновка с помощью диспетчера `BoxLayout`

Листинг 9.13. Содержимое файла `BoxLayoutText.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class BoxLayoutTest
{
   public static void main(String[] args)
```

```
   {
      BoxLayoutFrame frame = new BoxLayoutFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм, использующий диспетчер BoxLayout для размещения
   компонентов.
*/
class BoxLayoutFrame extends JFrame
{
   public BoxLayoutFrame()
   {
      setTitle("BoxLayoutTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Создание верхнего горизонтального блока.

      JLabel label1 = new JLabel("Name:");
      JTextField textField1 = new JTextField(10);
      textField1.setMaximumSize(textField1.getPreferredSize());

      Box hbox1 = Box.createHorizontalBox();
      hbox1.add(label1);
      // Разделение элементов распоркой размером 10 пикселей.
      hbox1.add(Box.createHorizontalStrut(10));
      hbox1.add(textField1);

      // Создание среднего горизонтального блока.

      JLabel label2 = new JLabel("Password:");
      JTextField textField2 = new JTextField(10);
      textField2.setMaximumSize(textField2.getPreferredSize());


      Box hbox2 = Box.createHorizontalBox();
      hbox2.add(label2);
      // Разделение элементов распоркой размером 10 пикселей.
      hbox2.add(Box.createHorizontalStrut(10));
      hbox2.add(textField2);

      // Создание нижнего горизонтального блока.

      JButton button1 = new JButton("OK");
      JButton button2 = new JButton("Cancel");

      Box hbox3 = Box.createHorizontalBox();
      hbox3.add(button1);
      // Для разделения кнопок используется склейка.
      hbox3.add(Box.createGlue());
      hbox3.add(button2);

      // Включение трех горизонтальных блоков в вертикальный блок.
```

```
        Box vbox = Box.createVerticalBox();
        vbox.add(hbox1);
        vbox.add(hbox2);
        vbox.add(Box.createGlue());
        vbox.add(hbox3);

        add(vbox, BorderLayout.CENTER);
    }

    public static final int DEFAULT_WIDTH = 200;
    public static final int DEFAULT_HEIGHT = 200;
}
```

javax.swing.Box 1.2

- `static Box createHorizontalBox()`
- `static Box createVerticalBox()`

Создает контейнер, содержимое которого упорядочено в горизонтальном или вертикальном направлении.

- `static Component createHorizontalGlue()`
- `static Component createVerticalGlue()`
- `static Component createGlue()`

Создает невидимый компонент, который может неограниченно расширяться в горизонтальном, вертикальном или в обоих направлениях.

- `static Component createHorizontalStrut(int width)`
- *`static Component createVerticalStrut(int height)`*
- `static Component createRigidArea(Dimension d)`

Создает невидимый компонент, с фиксированной шириной, или высотой, или с обоими фиксированными размерами.

java.awt.Component 1.0

- `float getAlignmentX()` **1.1**
- `float getAlignmentY()` **1.1**

Возвращает значение, определяющее выравнивание компонента по оси *x* или *y*. Значение 0 означает выравнивание по верхней либо левой границе, а 1 — по нижней либо правой границе.

Диспетчер компоновки GridBagLayout

`GridBagLayout` — прародитель всех диспетчеров компоновки. Его можно рассматривать как диспетчер `GridLayout` ограничений, т.е. строки и столбцы могут иметь переменный размер. Для того чтобы разместить более крупный компонент, который не помещается в одну ячейку, несколько смежных ячеек можно объединить в одну. (Многие текстовые редакторы, например HTML, предоставляют такие же возможности при редактировании таблиц: заполнение начинается с обычной сетки, а при необходимости некоторые ее ячейки сливаются в одну.) Компоненты не обязаны заполнять всю ячейку, поэтому можно задать выравнивание внутри ячеек.

Учтите, что, используя диспетчер `GridBagLayout`, вы можете столкнуться с существенными трудностями. Зато он обеспечивает наибольшую гибкость по сравнению с другими диспетчерами и может решать гораздо более широкий круг задач. Как вы помните, цель диспетчеров компоновки – сохранить расположение компонентов независимо от операционной системы, размеров элементов и особенностей шрифта.

В описании класса `BoxLayout` говорится: "Использование вложенных панелей, в которых компоненты упорядочены по вертикали либо по горизонтали, позволяет гораздо более простыми средствами достичь того же результата, который дает диспетчер компоновки `GridBagLayout`". Однако достоинства `BoxLayout` явно преувеличены. Для того чтобы выровнять компоненты, приходится использовать сложные сочетания блоков, распорок и склеек. Во многих случаях удобнее применять класс `GridBagLayout`.

Рассмотрим диалоговое окно, показанное на рис. 9.35. Оно содержит следующие компоненты.

- Два раскрывающихся списка, предназначенных для выбора начертания и размера шрифта.
- Метки указанных раскрывающихся списков.
- Два флажка опций для выбора полужирного шрифта и курсива.
- Текстовая область для отображения строки-примера.

Теперь разделим диалоговое окно на ячейки так, как показано на рис. 9.36. (Строки и столбцы не обязаны иметь одинаковый размер.) Как видим, каждый флажок занимает два столбца, а текстовая область – четыре строки.

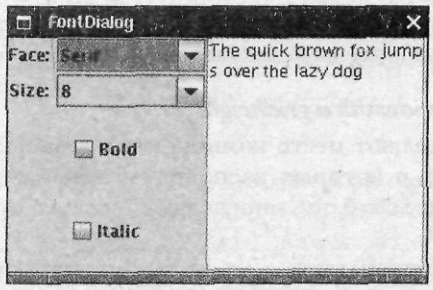


Рис. 9.35. Диалоговое окно, предназначенное для выбора шрифта

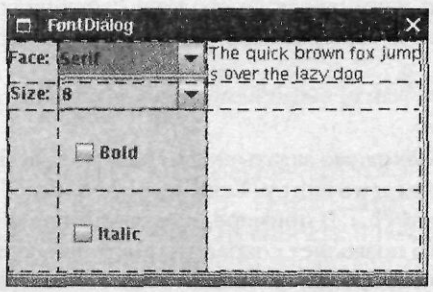


Рис. 9.36. Сетка диалогового окна, используемая при разработке

Чтобы описать эту схему в терминах диспетчера `GridBagLayout`, нужно выполнить ряд непростых действий.

1. Создать объект `GridBagLayout`. Конструктору не следует знать, сколько строк и столбцов содержит сетка. Позднее он попытается угадать эти параметры по информации, которую он получит от вас.
2. Связать объект `GridLayout` с контейнером.
3. Для каждого компонента создать объект `GridBagConstraints`. Этот объект будет задавать расположение компонентов.
4. Добавить компонент с ограничениями с помощью следующего выражения:

```
add(Component, constraints);
```

Ниже приведен соответствующий фрагмент кода. (Ограничения будут детально рассмотрены в дальнейшем — пока не следует беспокоиться о том, что они собой представляют.)

```
GridBagLayout layout = new GridBagLayout();
panel.setLayout(layout);
GridBagConstraints constraints = new GridBagConstraints();
constraints.weightx = 100;
constraints.weighty = 100;
constraints.gridx = 0;
constraints.gridy = 2;
constraints.gridwidth = 2;
constraints.gridheight = 1;
panel.add(style, bold);
```

Главная задача заключается в том, как задать состояние объекта класса `GridBagConstrains`. Наиболее важные сведения об этом объекте приводятся ниже.

Параметры gridx, gridy, gridwidth и gridheight

Эти параметры определяют место компонента в сетке. Значения `gridx` и `gridy` задают столбец и строку, в которых расположен левый верхний угол компонента. Значения `gridwidth` и `gridheight` определяют, сколько строк и столбцов занимает данный компонент.

Координаты сетки отсчитываются от нуля. В частности, выражения `gridx=0` и `gridy=0` означают левый верхний угол.

Например, текстовая область в нашем примере имеет параметры `gridx=2` и `gridy=0`, поскольку она начинается со столбца, имеющего номер 2 (т.е. третьего). Поскольку текстовая область занимает один столбец и четыре строки, `gridwidth=1`, а `gridheight=4`.

Весовые поля

Для каждой области необходимо задать *весовые поля* (weight field). В нашем примере им соответствуют поля `weightx` и `weighty`. Если вес равен нулю, то область всегда сохраняет свои первоначальные размеры. В примере, показанном на рис. 9.35, поле `weightx` для метки задано равным 0. Это позволяет сохранять постоянную ширину метки при изменении размера окна. С другой стороны, если задать вес всех областей равным 0, то контейнер разместит компоненты в центре и пространство окна не будет заполнено.

Проблема, связанная с весовыми параметрами, заключается в том, что они являются характеристиками строк и столбцов, а не отдельных ячеек. Несмотря на это, их нужно задавать для ячеек, поскольку диспетчер `GridBagLayout` не различает отдельные строки и столбцы. В качестве веса строки или столбца принимается максимальный из весов содержащихся в них ячеек. Таким образом, если нужно, чтобы строки и столбцы сохраняли фиксированный размер, необходимо задать вес всех компонентов в этих строках и столбцах равным нулю.

Обратите внимание на то, что вес на самом деле не позволяет определить относительные размеры столбцов. Он лишь указывает, какая часть "свободного" пространства должна быть выделена каждой области, если контейнер превышает рекомендуемый размер. Чтобы научиться правильно подбирать вес, надо иметь определенный опыт работы с данным диспетчером компоновки. Мы рекомендуем поступать так. Установите вес равным 100, затем запустите программу на выполнение и посмотрите, как будет выглядеть интерфейс. Для того чтобы определить, как выровнены строки и столбцы, попробуйте изменить размеры диалогового окна. Если окажется, что какая-то строка или столбец не должны увеличиваться, установите вес всех компонентов, находящихся в них, равным нулю.

Параметры fill и anchor

Если вы не хотите, чтобы компонент растягивался и заполнял все пространство, нужно установить параметр `fill`. Этот параметр принимает четыре возможных значения: `GridBagConstraints.NONE`, `GridBagConstraints.HORIZONTAL`, `GridBagConstraints.VERTICAL` и `GridBagConstraints.BOTH`.

Если компонент не должен заполнять собой все пространство, можно указать, к какой позиции вы хотели бы его привязать, задав параметр `anchor`. Этот параметр может принимать следующие значения: `GridBagConstraints.CENTER` (по умолчанию), `GridBagConstraints.NORTH`, `GridBagConstraints. NORTHEAST`, `GridBagConstraints.EAST` и т.д.

Выделение пустого пространства

Задав значение поля `insets` объекта `GridBagConstraints`, компонент можно окружить дополнительным пустым пространством. Для этого нужно задать значения полей `left`, `top`, `right` и `bottom` в объекте `Insets`. Этот процесс называется *внешним заполнением* (external padding).

Значения полей `ipadx` и `ipady` определяют *внутреннее заполнение* (internal padding). Эти значения добавляются к минимальным ширине и высоте компонента, гарантируя тем самым, что компонент никогда не уменьшится до своих минимальных размеров.

Альтернативные способы указания параметров gridx, gridy, gridwidth и gridheight

В документации по библиотеке AWT рекомендуется не задавать абсолютные значения параметров `gridx` и `gridy`, а вместо этого использовать константу `GridBagConstraints.RELATIVE`. Затем нужно, как обычно, слева направо строка за строкой включать компоненты в контейнер, управляемый диспетчером компоновки `GridBagLayout`.

Количество строк и столбцов, занятых ячейкой, нужно задавать с помощью полей gridheight и gridwidht. Исключение составляет компонент, занимающий *последнюю* строку или столбец. Для него задается не числовое значение, а специальная константа GridBagConstraints.REMAINDER. Это позволяет диспетчеру компоновки распознавать последние компоненты.

Данная схема вполне работоспособна. Однако кажется странным сначала скрывать от диспетчера компоновки информацию о фактическом расположении компонентов, а затем полагаться на то, что он сам угадает правильные параметры.

У читателя может сложиться впечатление, что пользоваться диспетчером GridBagLayout очень сложно. Однако, следуя приведенным ниже рекомендациям, удается размещать элементы, не затрачивая значительных усилий.

Компоновка с помощью диспетчера GridBagLayout

1. *Изобразите схему расположения компонентов на бумаге.*
2. Определите такую сетку, чтобы каждый компонент помещался в ячейке, а большие компоненты *могли занимать несколько ячеек.*
3. Пометьте строки и столбцы вашей сетки числами 0, 1, 2, 3,... . Теперь можно определять *значения параметров* gridx, gridy, gridwidth и gridheight.
4. Для каждого компонента решите, нужно ли, чтобы он заполнял ячейку в горизонтальном или вертикальном направлении? Если нет, как его выровнять внутри ячейки? Это определяют параметры fill и anchor.
5. Установите вес равным 100. Однако, если вы хотите, чтобы конкретная строка или столбец всегда сохраняли свой первоначальный размер, задайте параметры weightx или weighty всех компонентов, находящихся в этой строке или столбце, равными 0.
6. Напишите программу. Тщательно проверьте установки класса GridBagConstraints. Одно неверное ограничение может разрушить всю компоновку.
7. Скомпилируйте программу и запустите ее на выполнение.

Вспомогательный класс для работы с объектами GridBagConstraints

Наиболее трудоемкая задача при работе с диспетчером компоновки GridBagLayout — *это написание кода, устанавливающего ограничения. Многие программисты создают для* этой цели вспомогательные функции или небольшие вспомогательные классы. Код одного *из таких классов приведен после примера, демонстрирующего создание диалогового окна.* Характеристики вспомогательного класса приведены ниже.

- Имя класса — GBC (прописные буквы из имени GridBagConstraints).
- Данный класс является подклассом GridBagConstraints, поэтому константы можно указывать, используя более короткое имя, например GBC.EAST.
- Объект GBC используется при добавлении компонента, например:

```
add(component, new GBC(1, 2));
```

- Для установки наиболее часто используемых параметров, gridx и gridy или gridx, gridy, gridwidth и gridheight, предусмотрены два конструктора.

```
add(component, new GBC(1, 2, 1, 4));
```

- В данном классе предусмотрены удобные метки для установки полей, содержащих значения x и y.

```
add(component, new GBC(1, 2).setWeight(100, 100));
```

- Методы установки значений полей возвращают значение `this`, поэтому их можно объединять в цепочки.

```
add(component, new GBC(1, 2).setAnchor(GBC.EAST).setWeight(100, 100));
```

- Метод `setInsets()` создает объекты `Insets`. Так, например, чтобы создать пустое пространство размером в один пиксель, надо использовать следующее выражение:

```
add(component, new GBC(1, 2).setAnchor(GBC.EAST).setInsets(1));
```

Полный текст программы, реализующей диалоговое окно для выбора шрифта, содержится в листинге 9.14. Для добавления компонентов в контейнер, управляемый диспетчером `GridBagLayout`, используется приведенный ниже фрагмент кода.

```
add(faceLabel, new GBC(0, 0).setAnchor(GBC.EAST));
add(face, new GBC(1, 0).setFill(GBC.HORIZONTAL).
   setWeight(100, 0).setInsets(1));
add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
add(size, new GBC(1, 1).setFill(GBC.HORIZONTAL).
   setWeight(100, 0).setInsets(1));
add(bold, new GBC(0, 2, 2, 1).setAnchor(GBC.CENTER).
   setWeight(100, 100));
add(italic, new GBC(0, 3, 2, 1).
   setAnchor(GBC.CENTER).setWeight(100, 100));
add(sample, new GBC(2, 0, 1, 4).setFill(GBC.BOTH).
   setWeight(100, 100));
```

Понимая принцип использования ограничений при работе с диспетчером, `GridBagLayout`, в этом коде несложно разобраться.

В руководстве Sun, доступном по адресу `http://java.sun.com/docs/books/tutorial/uiswing/layout/gridbag.html`, предлагается использовать один и тот же объект `GridBagConstraints` для всех компонентов. На наш взгляд, получившийся в результате код труден для восприятия и при его написании легко допустить ошибку. Обратите внимание на демонстрационный код, расположенный в документе `http://java.sun.com/docs/books/tutorial/uiswing/events/containerlistener.html`. Действительно ли разработчик стремился растягивать кнопки по горизонтали, или он забыл отключить установку `BOTH` параметра `fill`?

Листинг 9.14. Содержимое файла `FontDialog.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class FontDialog
{
   public static void main(String[] args)
   {
```

```
      FontDialogFrame frame = new FontDialogFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм, в котором диспетчер компоновки GridBagLayout
   использован для выравнивания компонентов,
   предназначенных для выбора шрифта.
*/
class FontDialogFrame extends JFrame
{
   public FontDialogFrame()
   {
      setTitle("FontDialog");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      GridBagLayout layout = new GridBagLayout();
      setLayout(layout);

      ActionListener listener = new FontAction();

      // Создание компонентов.

      JLabel faceLabel = new JLabel("Face: ");

      face = new JComboBox(new String[]
         {
            "Serif", "SansSerif", "Monospaced",
            "Dialog", "DialogInput"
         });

      face.addActionListener(listener);

      JLabel sizeLabel = new JLabel("Size: ");

      size = new JComboBox(new String[]
         {
            "8", "10", "12", "15", "18", "24", "36", "48"
         });

      size.addActionListener(listener);

      bold = new JCheckBox("Bold");
      bold.addActionListener(listener);

      italic = new JCheckBox("Italic");
      italic.addActionListener(listener);

      sample = new JTextArea();
      sample.setText(
         "The quick brown fox jumps over the lazy dog");
      sample.setEditable(false);
      sample.setLineWrap(true);
      sample.setBorder(BorderFactory.createEtchedBorder());
```

```
      // Размещение компонентов с помощью класса GBC.

      add(faceLabel, new GBC(0, 0).setAnchor(GBC.EAST));
      add(face, new GBC(1, 0).setFill(GBC.HORIZONTAL).
         setWeight(100, 0).setInsets(1));
      add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
      add(size, new GBC(1, 1).setFill(GBC.HORIZONTAL).
         setWeight(100, 0).setInsets(1));
      add(bold, new GBC(0, 2, 2, 1).setAnchor(GBC.CENTER).
         setWeight(100, 100));
      add(italic, new GBC(0, 3, 2, 1).
         setAnchor(GBC.CENTER).setWeight(100, 100));
      add(sample, new GBC(2, 0, 1, 4).
         setFill(GBC.BOTH).setWeight(100, 100));
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   private JComboBox face;
   private JComboBox size;
   private JCheckBox bold;
   private JCheckBox italic;
   private JTextArea sample;

   /**
      Обработчик, изменяющий текст строки-примера.
   */
   private class FontAction implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         String fontFace = (String) face.getSelectedItem();
         int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
            + (italic.isSelected() ? Font.ITALIC : 0);
         int fontSize = Integer.parseInt(
           (String) size.getSelectedItem());
         Font font = new Font(fontFace, fontStyle, fontSize);
         sample.setFont(font);
         sample.repaint();
      }
   }
}
```

В листинге 9.15 показан код вспомогательного класса GBC.

Листинг 9.15. Содержимое файла GBC.java

```
/*
GBC - вспомогательный класс для работы с
диспетчером компоновки GridBagLayout

Copyright (C) 2002 Cay S. Horstmann (http://horstmann.com)

Данная программа распространяется свободно; по мере необходимости
```

```
вы можете распространять или модифицировать ее в рамках
второй или последующих версий GNU General Public License
Free Software Foundation.

Автор надеется, что эта программа будет полезна разработчикам,
но не дает никаких гарантий, в том числе гарантии пригодности
для определенной цели. Детали можно уточнить, ознакомившись с
GNU General Public License.

*/

import java.awt.*;

/**
   Данный класс упрощает работу с GridBagConstraints.
*/
public class GBC extends GridBagConstraints
{
   /**
      Создает объект GBC, определяя позиции gridx и gridy.
      Значения остальных параметров принимаются по умолчанию.
      @param gridx Позиция gridx
      @param gridy Позиция gridy
   */
   public GBC(int gridx, int gridy)
   {
      this.gridx = gridx;
      this.gridy = gridy;
   }

   /**
      Создает объект GBC, определяя gridx, gridy,
      gridwidth и gridheight. Значения остальных параметров
      принимаются по умолчанию.
      @param gridx Позиция gridx
      @param gridy Позиция gridy
      @param gridwidth Расширение ячейки в направлении x
      @param gridheight Расширение ячейки в направлении y
   */
   public GBC(int gridx, int gridy, int gridwidth, int gridheight)
   {
      this.gridx = gridx;
      this.gridy = gridy;
      this.gridwidth = gridwidth;
      this.gridheight = gridheight;
   }

   /**
      Устанавливает параметр anchor.
      @param anchor Значение параметра
      @return Объект this, пригодный для дальнейшей модификации
   */
   public GBC setAnchor(int anchor)
   {
      this.anchor = anchor;
      return this;
```

```
}

/**
   Устанавливает параметр fill.
   @param fill Значение параметра
   @return Объект this, пригодный для дальнейшей модификации
*/
public GBC setFill(int fill)
{
   this.fill = fill;
   return this;
}

/**
   Устанавливает веса ячейки.
   @param weightx Вес в направлении x
   @param weighty Вес в направлении y
   @return Объект this, пригодный для дальнейшей модификации
*/
public GBC setWeight(double weightx, double weighty)
{
   this.weightx = weightx;
   this.weighty = weighty;
   return this;
}

/**
   Устанавливает размеры свободного пространства для ячейки.
   @param distance Размеры по всем направлениям
   @return Объект this, пригодный для дальнейшей модификации
*/
public GBC setInsets(int distance)
{
   this.insets = new Insets(distance, distance,
      distance, distance);
   return this;
}

/**
   Устанавливает размеры свободного пространства для ячейки.
   @param top Размер верхней части свободного пространства
   @param left Размер левой части свободного пространства
   @param bottom Размер нижней части свободного пространства
   @param right Размер правой части свободного пространства
   @return Объект this, пригодный для дальнейшей модификации
*/
public GBC setInsets(int top, int left, int bottom, int right)
{
   this.insets = new Insets(top, left, bottom, right);
   return this;
}

/**
   Устанавливает внутреннее заполнение.
   @param ipadx Внутреннее заполнение в направлении x
   @param ipady Внутреннее заполнение в направлении y
```

```
        @return Объект this, пригодный для дальнейшей модификации
    */
    public GBC setIpad(int ipadx, int ipady)
    {
        this.ipadx = ipadx;
        this.ipady = ipady;
        return this;
    }
}
```

Некоторые средства создания графического пользовательского интерфейса допускают визуальную установку ограничений. Диалоговое окно NetBeans, предназначенное для конфигурирования, показано на рис. 9.37.

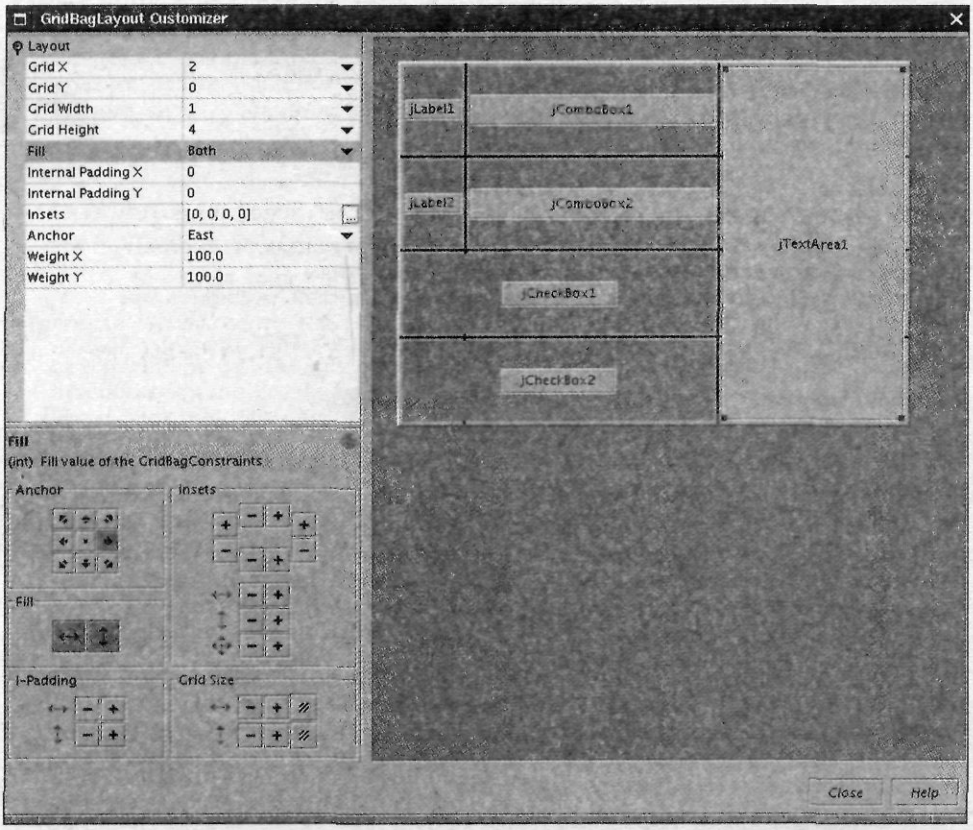


Рис. 9.37. Установка ограничений в NetBeans

java.awt.GridConstraints 1.0

- `int gridx, gridy`

  Задают столбец и строку для размещения ячейки. По умолчанию принимаются нулевые значения.

- `int gridwidth, gridheight`

  Задают количество столбцов и строк, занимаемых ячейкой. По умолчанию принимаются значения, равные 1.

- `double weightx, weighty`

  Определяют способность ячейки увеличиваться в размерах. По умолчанию принимается значение 0.

- `int anchor`

  Задает вид выравнивания компонента внутри ячейки. Вы можете указывать константы, определяющие абсолютную позицию:

NORTHWEST	NORTH	NORTHEAST
WEST	CENTER	EAST
SOUTHWEST	SOUTH	SOUTHEAST

  либо значения, не зависящие от ориентации:

FIRST_LINE_START	LINE_START	FIRST_LINE_END
PAGE_START	CENTER	PAGE_END
LAST_LINE_START	LINE_END	LAST_LINE_END

  Вторая группа констант удобна в том случае, если приложение адаптируется для языков, в которых символы следуют справа налево или сверху вниз.

- `int fill`

  Задает способ заполнения компонентом ячейки. Допустимы значения NONE, BOTH, HORIZONTAL и VERRTICAL. По умолчанию принимается NONE.

- `int ipadx, ipady`

  Задает внутреннее заполнение вокруг компонента. По умолчанию принимается значение 0.

- `Insets insets`

  Задает внешнее заполнение вокруг границ ячейки. По умолчанию заполнение отсутствует.

- `GridBagConstraints(int gridx, int gridy, int grigwidth, int gridheight, double weightx, double weighty, int anchor, int fill, Insets insets, int ipadx, int ipady)`

  Создает объект GridBagConstraints, заполняя все его поля указанными значениями. Компания Sun рекомендует использовать этот конструктор только в программах автоматической генерации интерфейса, поскольку полученный в результате код труден для восприятия.

Диспетчер компоновки SpringLayout

С момента появления класса `GridBagLayout` программисты неоднократно обращались к разработчикам языка Java с просьбой создать такой же гибкий, но более простой в использовании диспетчер компоновки. В конце концов в пакете JDK 1.4 появился класс `SpringLayout`. В этом разделе мы попробуем определить, соответствует ли он нашим ожиданиям.

При использовании данного диспетчера с каждым компонентом связываются так называемые *пружины* (spring). Пружина — это средство определения положения компонента на экране. Как показано на рис. 9.38, каждая пружина имеет следующие параметры.

- Минимальное значение.
- Рекомендуемое значение.
- Максимальное значение.
- Фактическое значение.

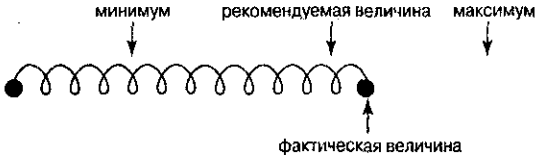


Рис. 9.38. Пружина, *связываемая с компонентом*

В классе, описывающем пружину, определена операция *суммирования*, получающая две пружины в качестве параметров и создающая новую пружину, объединяющую их свойства. Если в одной строке помещается большое количество компонентов, с этой строкой связывается несколько пружин, результат суммирования которых заполняет весь контейнер (рис. 9.39). Суммарная пружина сжимается или растягивается так, чтобы ее длина была равна размеру контейнера. Растяжение или сжатие передается каждой пружине, что приводит к фиксации компонентов. (Если вас заинтересовали детали вычислений, обратитесь к описанию класса `Spring` в документации по API.)

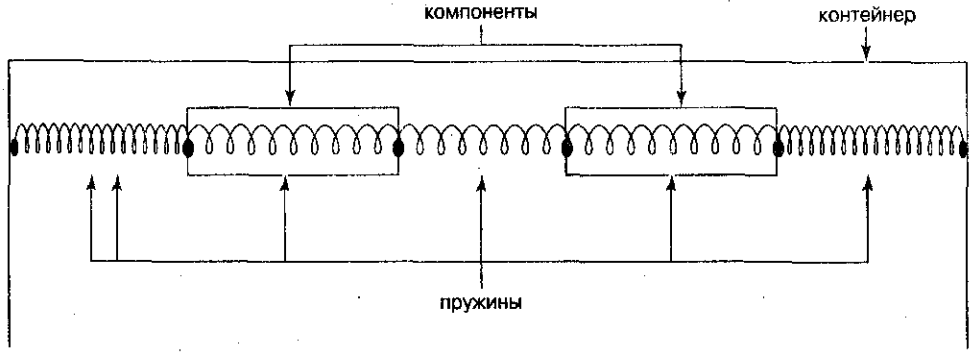


Рис. 9.39. Результат операции суммирования над пружинами

Рассмотрим простой пример. Допустим, нам нужно разместить горизонтально три кнопки.

```
JButton b1 = new JButton("Yellow");
JButton b2 = new JButton("Blue");
JButton b3 = new JButton("Red");
```

Сначала зададим SpringLayout в качестве диспетчера компоновки для фрейма и добавим компоненты.

```
SpringLayout layout = new SpringLayout();
panel.setLayout(layout);
panel.add(b1);
panel.add(b2);
panel.add(b3);
```

Теперь создадим пружину, способную сжиматься в широких пределах. Статический метод Spring.constant() создает пружину с заданными минимальным, рекомендуемым и максимальным значениями.

```
Spring s = Spring.constant(0, 10000, 10000);
```

Затем вставим одну из пружин между левой стороной контейнера и левой стороной кнопки b1.

```
layout.putConstraint(SpringLayout.WEST,
   b1, s, SpringLayout.WEST, panel);
```

Метод putConstraint() добавляет указанную пружину так, что ее конец находится в точке, определяемой первой группой параметров (в нашем примере это левая сторона кнопки b1), а начало – в точке, определяемой второй группой параметров (левая сторона контейнера).

Затем между компонентами вставляются остальные пружины.

```
layout.putConstraint(SpringLayout.WEST,
   b2, s, SpringLayout.EAST, b1);
layout.putConstraint(SpringLayout.WEST,
   b3, s, SpringLayout.EAST, b2);
```

В заключение вставляется пружина, соединяющая кнопку b3 с правой стороной контейнера.

```
layout.putConstraint(SpringLayout.EAST,
   panel, s, SpringLayout.EAST, b3);
```

В результате на панели окажутся четыре пружины, сжатые до одинаковых размеров, а между кнопками будет установлено одинаковое расстояние (рис. 9.40).

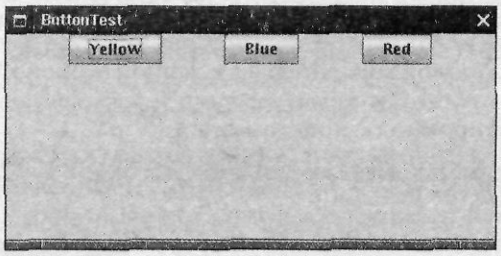


Рис. 9.40. Между кнопками установлено одинаковое расстояние

Расстояние между кнопками может варьироваться. Допустим, что оно должно быть заранее задано. Для этого можно использовать *распорку* (strut) — пружину, которая не способна сжиматься и растягиваться. Такую пружину можно создать с помощью одного из вариантов метода `Spring.constant`.

```
Spring strut = Spring.constant(10);
```

Если между кнопками вставить две распорки, а между сторонами контейнера и кнопками — две пружины, получится группа кнопок, расположенных симметрично относительно центра контейнера (рис. 9.41).

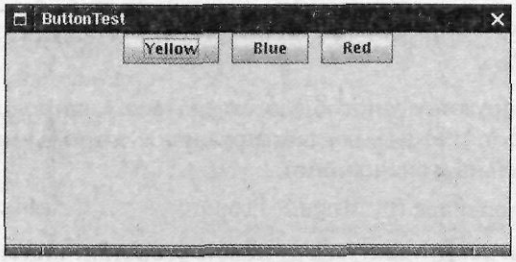


Рис. 9.41. Использование пружин и распорок

Разумеется, для выполнения такой простой задачи можно было бы обойтись и без пружин. Поэтому рассмотрим более сложный случай: часть диалогового окна для выбора шрифта. В окне расположены два раскрывающихся списка, снабженных метками. Мы хотели бы, чтобы левые стороны обоих списков начинались после *более длинной* метки (рис. 9.42).

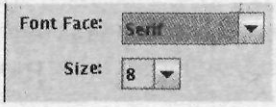


Рис. 9.42. Выравнивание по столбцам

Для этого нужна другая операция над пружинами. С помощью метода `Spring.max()` можно создать *максимальную* из двух пружин. Результатом этой операции является пружина, которая имеет наибольшую длину.

Определим, у какой из меток правая сторона располагается правее.

```
Spring labelsEast = Spring.max(
   layout.getConstraint(SpringLayout.EAST, faceLabel),
   layout.getConstraint(SpringLayout.EAST, sizeLabel));
```

Обратите внимание на то, что метод `getConstraint()` создает пружину, вставленную между левой стороной контейнера и заданной стороной компонента.

Добавим распорку так, чтобы между метками и раскрывающимися списками оставалось небольшое фиксированное пространство.

```
Spring comboWest = Spring.sum(labelsEast, strut);
```

Теперь соединим эти пружины с левыми сторонами обоих раскрывающихся списков. Каждая пружина начинается на левой сторое контейнера, поскольку именно с ней связана пружина `labelsEast`.

```
layout.putConstraint(SpringLayout.WEST, face,
   combosWest,SpringLayout.WEST, panel);
layout.putConstraint(SpringLayout.WEST, size,
   combosWest,SpringLayout.WEST, panel);
```

Теперь оба раскрывающихся списка выровнены, поскольку они связаны с одной и той же пружиной.

Однако остался один недостаток. Мы хотели бы, чтобы метки были выровнены по правому краю. Для этого нужно лучше разобраться в способах соединениях пружин.

Рассмотрим подробнее горизонтальные пружины. Вертикальные пружины устроены точно так же. На рис. 9.43 показаны три способа соединения пружин с компонентами.

- Соединение с левой стороны компонента и с левой стороной другого компонента.
- Вставка внутрь компонента.
- *Соединение с левой стороной компонента и с правой стороной другого компонента.*

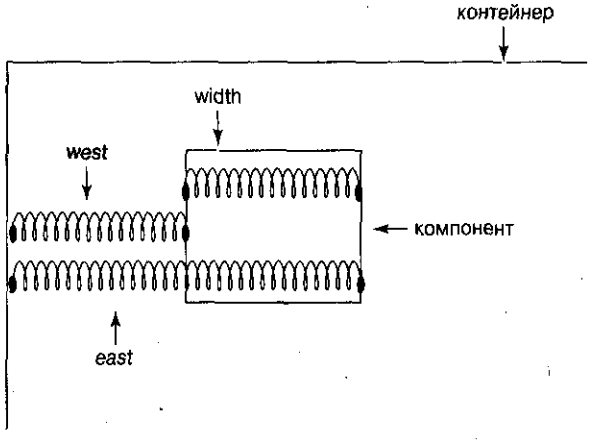


Рис. 9.43. Соединение горизонтальных пружин с компонентом

Эти три пружины можно создать следующим образом

```
Spring west = layout.getConstraints(component).getX();
Spring width = layout.getConstraints(component).getWidth();
Spring east = layout.getConstraint(SpringLayout.EAST, component);
```

Метод `getConstraints()` создает объект `SpringLayout.Constraints`. Этот *объект можно считать прямоугольником, координаты, ширина и высота которого* задаются пружинами, а не числами. Метод `getConstraint()` создает отдельную пружину, достигающую одной из четырех сторон компонента. Например, пружину `west` можно создать с помощью следующего выражения:

```
Spring west = layout.getConstraint(SpringLayout.WEST,
  component);
```

Разумеется, все три пружины связаны между собой: сумма пружин `west` и `width` должна быть равна пружине `east`.

Если ограничения для компонента заданы заранее, пружиной `width` считается пружина, параметрами которой являются минимальная, рекомендуемая и максимальная ширина компонента. Пружина `west` устанавливается равной 0.

Если не установить значение пружины `west` (или `north`), то смещение компонента относительно контейнера будет нулевым.

Если компонент соединен с двумя пружинами, а мы добавляем третью, он становится излишне ограниченным. В табл. 9.3 указаны пружины, параметры которых должны быть вычислены заново.

Таблица 9.3. Добавление пружины к излишне ограниченному компоненту

Добавляемая пружина	Удаляемая пружина	Замена
`west`	`width`	`east - west`
`width`	`east`	`west + width`
`east`	`west`	`east - width`

Разность двух пружин может быть не очевидной, но вполне осмысленной. В языке Java нет метода для вычитания пружин. Если нужно вычислить разность двух пружин, следует использовать следующее выражение:

```
Spring.sum(s, Spring.minus(t));
```

Теперь нам известно о пружинах достаточно, чтобы решить проблему выравнивания правых сторон компонентов. Вычислим максимальную ширину обоих меток. Затем присвоим *правым* пружинам обоих меток максимальное значение. Как видно из табл. 9.3, ширина метки не изменится, величина левой пружины будет вычислена заново, а правые стороны меток выровняются.

```
Spring labelsEast = Spring.sum(strut,
   Spring.max(layout.getConstraints(faceLabel).getWidth(),
   Spring.max(layout.getConstraints(sizeLabel).getWidth()));
layout.putConstraint(SpringLayout.EAST, faceLabel,
   labelsEast, SpringLayout.WEST, panel);
layout.putConstraint(SpringLayout.EAST, sizeLabel,
   labelsEast, SpringLayout.WEST, panel);
```

В листинге 9.16 показано, как скомпоновать диалоговое окно для выбора шрифта с помощью диспетчера `SpringLayout`. Если внимательно проанализировать код, вы, возможно, согласитесь, что такая компоновка менее очевидна, чем компоновка с помощью диспетчера `GridBagLayout`. Мы надеемся, что впоследствии все же появятся инструментальные средства, облегчающие работу с `SpringLayout`. Пока этого не произошло, мы рекомендуем использовать диспетчер `GridBagLayout`.

Листинг 9.16. Содержимое файла `SpringLayoutTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
import javax.swing.event.*;

public class SpringLayoutTest
{
   public static void main(String[] args)
   {
      FontDialogFrame frame = new FontDialogFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм, в котором для размещения компонетов
   используется диспетчер компоновки SpringLayout.
*/
class FontDialogFrame extends JFrame
{
   public FontDialogFrame()
   {
      setTitle("FontDialog");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      JPanel panel = new JPanel();
      SpringLayout layout = new SpringLayout();
      panel.setLayout(layout);

      ActionListener listener = new FontAction();

      // Создание компонентов

      JLabel faceLabel = new JLabel("Font Face: ");

      face = new JComboBox(new String[]
         {
            "Serif", "SansSerif", "Monospaced",
            "Dialog", "DialogInput"
         });

      face.addActionListener(listener);

      JLabel sizeLabel = new JLabel("Size: ");

      size = new JComboBox(new String[]
         {
            "8", "10", "12", "15", "18", "24", "36", "48"
         });

      size.addActionListener(listener);

      bold = new JCheckBox("Bold");
      bold.addActionListener(listener);

      italic = new JCheckBox("Italic");
      italic.addActionListener(listener);
```

```
sample = new JTextArea();
sample.setText(
   "The quick brown fox jumps over the lazy dog");
sample.setEditable(false);
sample.setLineWrap(true);
sample.setBorder(BorderFactory.createEtchedBorder());

panel.add(faceLabel);
panel.add(sizeLabel);
panel.add(face);
panel.add(size);
panel.add(bold);
panel.add(italic);
panel.add(sample);

// Добавление пружин для размещения компонентов.
Spring strut = Spring.constant(10);

Spring labelsEast = Spring.sum(strut,
   Spring.max(
      layout.getConstraints(faceLabel).getWidth(),
      layout.getConstraints(sizeLabel).getWidth()));

layout.putConstraint(SpringLayout.EAST, faceLabel,
   labelsEast, SpringLayout.WEST, panel);
layout.putConstraint(SpringLayout.EAST, sizeLabel,
   labelsEast, SpringLayout.WEST, panel);

layout.putConstraint(SpringLayout.NORTH, faceLabel,
   strut, SpringLayout.NORTH, panel);
layout.putConstraint(SpringLayout.NORTH, face, strut,
   SpringLayout.NORTH, panel);

Spring secondRowNorth = Spring.sum(strut,
   Spring.max(
      layout.getConstraint(SpringLayout.SOUTH, faceLabel),
      layout.getConstraint(SpringLayout.SOUTH, face)));

layout.putConstraint(SpringLayout.NORTH, sizeLabel,
   secondRowNorth, SpringLayout.NORTH, panel);
layout.putConstraint(SpringLayout.NORTH, size,
   secondRowNorth, SpringLayout.NORTH, panel);

layout.putConstraint(SpringLayout.WEST, face, strut,
   SpringLayout.EAST, faceLabel);
layout.putConstraint(SpringLayout.WEST, size, strut,
   SpringLayout.EAST, sizeLabel);

layout.putConstraint(SpringLayout.WEST, bold, strut,
   SpringLayout.WEST, panel);
layout.putConstraint(SpringLayout.WEST, italic, strut,
   SpringLayout.WEST, panel);

Spring s = Spring.constant(10, 10000, 10000);

Spring thirdRowNorth = Spring.sum(s,
```

```
            Spring.max(
               layout.getConstraint(SpringLayout.SOUTH, sizeLabel),
               layout.getConstraint(SpringLayout.SOUTH, size)));

         layout.putConstraint(SpringLayout.NORTH, bold,
               thirdRowNorth, SpringLayout.NORTH, panel);
         layout.putConstraint(SpringLayout.NORTH, italic, s,
               SpringLayout.SOUTH, bold);
         layout.putConstraint(SpringLayout.SOUTH, panel, s,
               SpringLayout.SOUTH, italic);

         Spring secondColumnWest = Spring.sum(strut,
            Spring.max(
               layout.getConstraint(SpringLayout.EAST, face),
               layout.getConstraint(SpringLayout.EAST, size)));

         layout.putConstraint(SpringLayout.WEST, sample,
            secondColumnWest, SpringLayout.WEST, panel);
         layout.putConstraint(SpringLayout.SOUTH, sample,
            Spring.minus(strut), SpringLayout.SOUTH, panel);
         layout.putConstraint(SpringLayout.NORTH, sample,
            strut, SpringLayout.NORTH, panel);
         layout.putConstraint(SpringLayout.EAST, panel, strut,
            SpringLayout.EAST, sample);

         add(panel);
      }

      public static final int DEFAULT_WIDTH = 400;
      public static final int DEFAULT_HEIGHT = 200;

      private JComboBox face;
      private JComboBox size;
      private JCheckBox bold;
      private JCheckBox italic;
      private JTextArea sample;

      /**
         Обработчик, изменяющий шрифт, которым отображается
         строка-образец.
      */
      private class FontAction implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            String fontFace = (String) face.getSelectedItem();
            int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
               + (italic.isSelected() ? Font.ITALIC : 0);
            int fontSize = Integer.parseInt(
               (String) size.getSelectedItem());
            Font font = new Font(fontFace, fontStyle, fontSize);
            sample.setFont(font);
            sample.repaint();
         }
      }
}
```

API javax.swing.SpringLayout 1.4

- `SpringLayout getConstraints(Component c)`
  Возвращает ограничения для заданного компонента.

  Параметры `c` Один из компонентов или контейнер, управляемый диспетчером компоновки

- `void putConstraints(String endSide, Component end, Spring s, String startSide, Component start)`
- `void putConstraints(String endSide, Component end, int pad, String startSide, Component start)`

  Связывает указанную сторону компонента `end` с пружиной. Она получается путем сложения пружины `s` или распорки, имеющей размер `pad`, с пружиной, соединяющей левую сторону контейнера с указанной стороной контейнера `start`.

  Параметры

`endSide`, `startSide`	`WEST`, `EAST`, `NORTH` или `SOUTH`
`end`	Компонент, к которому добавляется пружина
`s`	Одна из слагаемых пружин
`pad`	Размер распорки
`start`	Компонент, с которым соединяется другая слагаемая пружина

API javax.swing.SpringLayout.Constraints 1.4

- `Constraints(Component c)` 5.0

  Создает объект `Constraints`, для которого позиция и ширина пружины соответствуют указанному компоненту.

- `Spring getX()`
- `Spring getY()`

  Возвращает пружину, соединяющую начало контейнера с левой или верхней стороной компонента.

- `Spring getWidth()`
- `Spring getHeight()`

  Возвращает горизонтальную или вертикальную пружину, вставленную в компонент.

- `Spring getConstraint(String side)`
- `void setConstarint(String edge, Spring s)`

  Возвращает или устанавливает пружину, соединяющую начало контейнера с заданной стороной компонента.

  Параметры

`side`	Одна из констант `WEST`, `EAST`, `NORTH` или `SOUTH`, определенных в классе `SpringLayout`
`s`	Устанавливаемая пружина

javax.swing.Spring 1.4

- `static Spring constant(int preferred)`

  Создает распорку. В качестве параметра указывается рекомендуемый размер. Минимальный и максимальный размеры задаются равными рекомендуемому.

- `static Spring constant(int minimum, int preferred, int maximum)`

  Создает пружину с заданными минимальным, рекомендуемым и максимальным размерами.

- `static Spring sum(Spring s, Spring t)`

  Возвращает сумму пружин `s` и `t`.

- `static Spring max(Spring s, Spring t)`

  Возвращает максимальную из двух пружин, `s` и `t`.

- `static Spring minus(Spring s)`

  Возвращает пружину, обратную пружине `s`.

- `static Spring scale(Spring s, float factor)` 5.0

  Умножает минимальный, рекомендуемый и максимальный размеры пружины `s` на заданный множитель. Если множитель отрицательный, умножается пружина, обратная пружине `s`.

- `static Spring width(Component c)` 5.0
- `static Spring height(Component c)` 5.0

  Возвращает пружину, минимальный, рекомендуемый и максимальный размер которой совпадает с минимальной, рекомендуемой и максимальной шириной или высотой заданного компонента.

- `int getMinimumValue()`
- `int getPreferred()`
- `int getMaximumValue()`

  Возвращает минимальный, рекомендуемый и максимальный размер пружины.

- `int getValue()`
- `void setValue(int newValue)`

  Возвращает или устанавливает размер пружины. При задании размера составной пружины одновременно задаются размеры компонентов.

Компоновка без использования диспетчера

Иногда бывает нужно просто поместить какой-то компонент в требуемую позицию, не прибегая к помощи диспетчера компоновки. Такой процесс иногда называют *абсолютным позиционированием* (absolute positioning). Если приложение задумывается как платформенно-независимое, такой подход не годится, но если вы хотите быстро создать прототип интерфейса, он вполне приемлем.

Чтобы разместить компонент в требуемой позиции, надо выполнить следующие действия.

1. Вместо диспетчера компоновки задать значение `null`.
2. Добавить компонент в контейнер.
3. Задать координаты и размер.

```
setLayout(null);
JButton ok = new JButton("OK");
add(ok);
ok.setBounds(10, 10, 30, 15);
```

java.awt.Component 1.0

- `void setBounds(int x, int y, int width, int height)`

  Перемещает компонент и изменяет его размеры.

Параметры	`x,y`	Координаты левого верхнего угла компонента
	`width, height`	Новые размеры компонента

Диспетчер компоновки, определяемый разработчиком

В принципе, можно разработать собственный класс `LayoutManager`, управляющий компонентами по-своему. Например, можно расставить все компоненты в контейнере по кругу. Разумеется, почти всегда это лишь напрасная трата времени и сил, но результат, как показывает рис. 9.44, может быть впечатляющим.

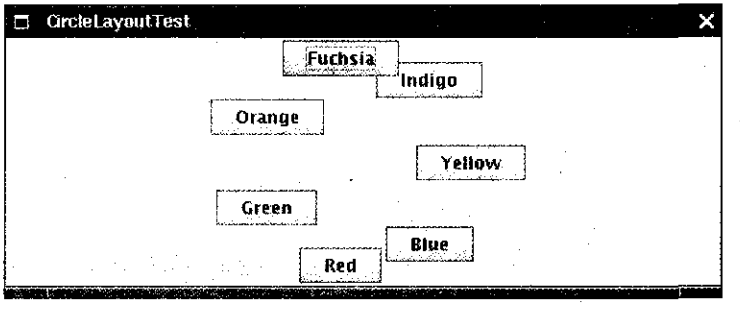


Рис. 9.44. Размещение компонентов по кругу

Если вы жить не можете без собственного диспетчера компоновки, попробуйте сделать его сами. Ваш класс должен реализовывать интерфейс `LayoutManager`. Вам придется переопределить следующие пять методов.

```
void addLayoutComponent(String s, Component c);
void removeLayoutComponent(Component c);
Dimension preferredLayoutSize(Container parent);
Dimension minimumLayoutSize(Container parent);
void layoutContainer(Container parent);
```

Первые два метода вызываются при добавлении или удалении компонента. Если никакой дополнительной информации о компоненте нет, можно оставить эти методы пустыми. Следующие два метода вычисляют объем пространства, необходимого при минимальных и рекомендуемых размерах компонентов. Обычно эти величины равны. Пятый метод вызывает `setBounds()` для всех компонентов и выполняет основную работу.

В библиотеке AWT есть второй интерфейс, который называется `LayoutManager2`. В нем содержатся 10 методов, а не 5. Главная особенность этого интерфейса состоит в том, что с его помощью можно использовать метод `add()` без каких-либо ограничений. Например, классы `BorderLayout` и `GridBagLayout` реализуют именно интерфейс `LayoutManager2`.

В листинге 9.17 приведен код программы, реализующей диспетчер компоновки `CircleLayout`, который размещает компоненты вдоль окружности внутри контейнера. Это довольно забавно и совершенно бесполезно.

Листинг 9.17. Содержимое файла `CircleLayoutTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CircleLayoutTest
{
   public static void main(String[] args)
   {
      CircleLayoutFrame frame = new CircleLayoutFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.pack();
      frame.setVisible(true);
   }
}

/**
   Фрейм, в котором кнопки размещаются вдоль окружности.
*/
class CircleLayoutFrame extends JFrame
{
   public CircleLayoutFrame()
   {
      setTitle("CircleLayoutTest");

      setLayout(new CircleLayout());
      add(new JButton("Yellow"));
      add(new JButton("Blue"));
      add(new JButton("Red"));
      add(new JButton("Green"));
      add(new JButton("Orange"));
      add(new JButton("Fuchsia"));
      add(new JButton("Indigo"));
   }
}
```

```
/**
   Диспетчер компоновки, обеспечивающий размещение
   компонентов вдоль окружности.
*/
class CircleLayout implements LayoutManager
{
   public void addLayoutComponent(String name, Component comp)
   {}

   public void removeLayoutComponent(Component comp)
   {}

   public void setSizes(Container parent)
   {
      if (sizesSet) return;
      int n = parent.getComponentCount();

      preferredWidth = 0;
      preferredHeight = 0;
      minWidth = 0;
      minHeight = 0;
      maxComponentWidth = 0;
      maxComponentHeight = 0;

      // Вычисление максимальной ширины и высоты компонента.
      // Рекомендуемый размер устанавливается
      // как сумма размеров.
      for (int i = 0; i < n; i++)
      {
         Component c = parent.getComponent(i);
         if (c.isVisible())
         {
            Dimension d = c.getPreferredSize();
            maxComponentWidth =
               Math.max(maxComponentWidth, d.width);
            maxComponentHeight =
               Math.max(maxComponentHeight, d.height);
            preferredWidth += d.width;
            preferredHeight += d.height;
         }
      }
      minWidth = preferredWidth / 2;
      minHeight = preferredHeight / 2;
      sizesSet = true;
   }

   public Dimension preferredLayoutSize(Container parent)
   {
      setSizes(parent);
      Insets insets = parent.getInsets();
      int width = preferredWidth + insets.left + insets.right;
      int height = preferredHeight + insets.top + insets.bottom;
      return new Dimension(width, height);
   }
```

```
public Dimension minimumLayoutSize(Container parent)
{
   setSizes(parent);
   Insets insets = parent.getInsets();
   int width = minWidth + insets.left + insets.right;
   int height = minHeight + insets.top + insets.bottom;
   return new Dimension(width, height);
}

public void layoutContainer(Container parent)
{
   setSizes(parent);

   // Определение центра окружности.

   Insets insets = parent.getInsets();
   int containerWidth =
      parent.getSize().width - insets.left - insets.right;
   int containerHeight =
      parent.getSize().height - insets.top - insets.bottom;

   int xcenter = insets.left + containerWidth / 2;
   int ycenter = insets.top + containerHeight / 2;

   // Определение радиуса окружности.

   int xradius = (containerWidth - maxComponentWidth) / 2;
   int yradius = (containerHeight - maxComponentHeight) / 2;
   int radius = Math.min(xradius, yradius);

   // Размещение компонентов по окружности.

   int n = parent.getComponentCount();
   for (int i = 0; i < n; i++)
   {
      Component c = parent.getComponent(i);
      if (c.isVisible())
      {
         double angle = 2 * Math.PI * i / n;

         // Центр компонента.
         int x = xcenter + (int)(Math.cos(angle) * radius);
         int y = ycenter + (int)(Math.sin(angle) * radius);

         // Компонент располагается так, чтобы его
         // центр находился в точке (x, y), а размеры
         // были равны рекомендуемым.
         Dimension d = c.getPreferredSize();
         c.setBounds(x - d.width / 2, y - d.height / 2,
            d.width, d.height);
      }
   }
}

private int minWidth = 0;
private int minHeight = 0;
```

```
   private int preferredWidth = 0;
   private int preferredHeight = 0;
   private boolean sizesSet = false;
   private int maxComponentWidth = 0;
   private int maxComponentHeight = 0;
}
```

java.awt.LayoutManager 1.0

- `void addLayoutComponent(String name, Component comp)`

  Добавляет компонент в схему компоновки.

  Параметры `name` Строка, задающая координаты компонента
  `comp` Добавляемый компонент

- `void removeLayoutComponent(Component comp)`

  Удаляет компонент из схемы компоновки.

  Параметр `comp` Удаляемый компонент

- `Dimension preferredLayoutSize(Container parent)`

  Возвращает рекомендуемые размеры контейнера, в котором выполняется компоновка.

  Параметр `parent` Контейнер, в котором выполняется компоновка

- `Dimension minimumLayoutSize(Container parent)`

  Возвращает минимальные размеры контейнера, в котором выполняется компоновка.

  Параметр `parent` Контейнер, в котором выполняется компоновка.

- `void layoputContainer(Container parent)`

  Выполняет компоновку внутри контейнера.

  Параметр `parent` Контейнер, в котором выполняется компоновка

Порядок обхода компонентов

Если в окне содержится много компонентов, нужно подумать о *порядке их обхода* (traversal order). Когда окно открывается в первый раз, фокус ввода находится на том компоненте, который считается первым при заданном порядке обхода. Каждый раз, когда пользователь нажимает клавишу <Tab>, фокус перемещается на следующий компонент. (Напомним, что с компонентом, обладающим фокусом ввода, можно работать с помощью клавиш. Например, на кнопке, владеющей фокусом, можно "щелкнуть" с помощью клавиши пробела.) Вы лично можете оставаться безразличным к такой возможности, однако есть много пользователей, которые предпочитают работать именно так. Порядок обхода элементов имеет большое значение при использовании речевого интерфейса.

В пакете Swing реализован вполне разумный порядок обхода: сначала — слева направо, затем – сверху вниз. Например, в диалоговом окне для выбора шрифта обход компонентов осуществлялся в следующем порядке (рис. 9.45).

1. Раскрывающийся список, предназначенный для выбора начертаний.
2. Текстовая область с фрагментом текста, используемым в качестве образца.
3. Раскрывающийся список, предназначенный для выбора размеров.
4. Флажок опции для установки полужирного шрифта.
5. Флажок опции для установки курсива.

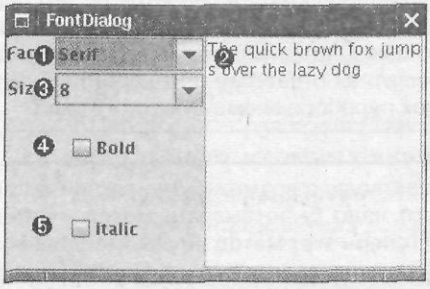


Рис. 9.45. Порядок обхода компонентов, принятый по умолчанию

В библиотеке AWT порядок обхода определялся очередностью, в которой компоненты вставлялись в контейнер. В библиотеке Swing порядок включения компонентов не имеет значения — учитывается только вид компоновки.

Если контейнеры вложены друг в друга, ситуация усложняется. Когда фокус передается вложенному контейнеру, его автоматически получает левый верхний компонент, а затем начинается обход остальных компонентов. В конце концов фокус передается компоненту, следующему за контейнером. Это обстоятельство можно с выгодой использовать, помещая группу связанных между собой элементов в другой контейнер, например панель.

При работе с JDK 1.4 можно указать, чтобы некоторый компонент при обходе не получал фокус ввода. Для этого следует вызывать метод

```
component.setFocusable(false);
```

В предыдущих версиях пакета JDK для этого требовалось переопределять метод `isFocusTraversable()`, но теперь этот метод не рекомендован к использованию.

Подводя итог, обратим внимание на две стандартные стратегии обхода, принятые в JDK 1.4.

- Приложения, использующие исключительно AWT, применяют политику `DefaultFocusTraversablePolicy`. Компоненты получают при обходе фокус ввода, если они являются видимыми, отображаются на экране, доступны и передача им фокуса не запрещена. Кроме того, требуется, чтобы их платформен-

но-зависимые эквиваленты также могли иметь фокус. Обход компонентов осуществляется в том же порядке, в котором они были вкючены в контейнер.

- Приложения, использующие библиотеку Swing, используют политику `LayoutFocusTraversablePolicy`. Компоненты получают фокус, если они являются видимыми, отображаются на экране, доступны и передача им фокуса не запрещена. Обход производится слева направо и сверху вниз. Однако вложенный контейнер создает новый "цикл обхода" — сначала перебираются содержащиеся в нем компоненты, а лишь затем фокус передается компоненту, следующему за контейнером.

Термин "цикл" может привести к недоразумениям. По достижении последнего элемента дочернего контейнера фокус передается не его первому элементу, а следующему компоненту родительского контейнера. В API реализованы настоящие циклы, включая поддержку клавиш, предназначенных для перехода вверх и вниз по иерархии циклов. Однако стандартная стратегия обхода не использует иерархических циклов, она реализует порядок перебора элементов "в глубину".

В пакете JDK 1.3 существовала возможность изменить порядок обхода, принятый по умолчанию. Для этого надо было вызвать метод `setNextFocusableComponent()` класса `JComponent`. Теперь этот метод не рекомендован к применению. Чтобы изменить порядок перебора компонентов, попробуйте сгруппировать связанные между собой компоненты на отдельных панелях так, чтобы они образовали циклы. Если этот способ не даст нужного результата, надо либо установить средства переупорядочения компонентов, либо полностью заменить стратегию обхода. Ни один из этих подходов не внушает доверия — детали можно найти в документации по API.

Диалоговые окна

До сих пор мы рассматривали только компоненты пользовательского интерфейса, которые находились в окне фрейма, создаваемого в приложении. Однако при создании приложений возникает необходимость в отдельных диалоговых окнах, которые отображались бы на экране и обеспечивали обмен информацией с пользователем.

Как и в большинстве оконных систем, в AWT различаются *модальные* и *немодальные* диалоговые окна. Модальные диалоговые окна не позволяют пользователю одновременно работать с другими окнами приложения. Такие окна необходимы, если пользователь должен ввести информацию, от которой зависит дальнейшая работа приложения. Например, при считывании файла пользователь должен сначала указать его имя. Только когда пользователь закроет модальное диалоговое окно, программа приступит к выполнению операции считывания.

Немодальное диалоговое окно позволяет пользователю одновременно вводить информацию как в нем, так и в других окнах приложения. Примером такого окна является панель инструментов. Она остается на месте сколь угодно долго, и пользователь может одновременно взаимодействовать как с ней, так и с другими окнами.

Начнем этот раздел с простейшего модального диалогового окна, содержащего строку сообщения. В пакете Swing есть удобный класс `JOptionPane`, позволяющий выводить на экран модальное диалоговое окно, не создавая специального кода для его поддержки. Далее мы расскажем, как можно реализовать свой собственный диалог. А в заключение мы покажем, как передавать информацию из приложения в диалоговое окно и обратно.

Этот раздел завершается анализом двух стандартных диалоговых окон, предназначенных для работы с файлами и установки цветов. Диалоговое окно для работы с файлами устроено достаточно сложно. Для работы с ним необходимо хорошо знать класс `JFileChooser` из библиотеки Swing. Для выбора цветов из палитры вам может пригодиться диалоговое окно `JColorChooser`.

Диалоговые окна для выбора вариантов

В библиотеке Swing содержится много готовых диалоговых окон, которые позволяют вводить небольшие объемы информации. В классе `JOptionPane` есть четыре статических метода.

`showMessageDialog`	Выводит на экран сообщение и ожидает, пока пользователь не щелкнет на кнопке OK
`showConfirmDialog`	Выводит на экран сообщение и ждет подтверждения (OK или Cancel)
`showOptionDialog`	Выводит на экран сообщение и предоставляет пользователю выбор из нескольких вариантов
`showInputDialog`	Выводит на экран сообщение и поле редактирования, в котором пользователь должен ввести информацию

На рис. 9.46 показано типичное диалоговое окно. Как видите, в нем содержатся следующие компоненты.

- Пиктограмма.
- Сообщение.
- Одна или несколько кнопок.

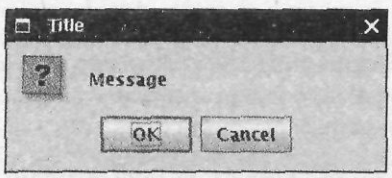


Рис. 9.46. Диалоговое окно, содержащее пиктограмму, сообщение и две кнопки

Диалоговое может содержать компонент для ввода данных. Этим компонентом может быть поле редактирования, в котором пользователь вводит произвольную строку, или раскрывающийся список, один из пунктов которого пользователь должен выбрать.

Компоновка диалоговых окон и выбор пиктограмм для стандартных сообщений зависит от стиля интерфейса.

Пиктограмма в левой части окна выбирается в зависимости от типа сообщения. Существуют пять типов сообщений.

```
ERROR_MESSAGE
INFORMATION_MESSAGE
WARNING_MESSAGE
QUESTION_MESSAGE
PLAIN_MESSAGE
```

Для сообщения PLAIN_MESSAGE пиктограмма не предусмотрена. Для каждого из типов диалоговых окон существует также метод, позволяющий использовать свою собственную пиктограмму.

С каждым из типов диалоговых окон можно связать определенное сообщение, которое может быть строкой, пиктограммой, компонентом пользовательского интерфейса или любым другим объектом. Ниже показано, как эти сообщения отображаются на экране.

`String`	Выводит строку.
`Icon`	Отображает пиктограмму.
`Component`	Отображает компонент.
`Object[]`	Выводит все объекты, отображая их один поверх другого.
Любой другой объект	Применяет метод `toString` и выводит на экран строку, полученную в результате этого преобразования.

Все эти возможности продемонстрированы в программе, приведенной в листинге 9.18.

Разумеется, в большинстве случаев на экран выводится именно строка сообщения. В то же время возможность отображать в диалоговом окне объекты класса `Component` обеспечивает большую гибкость.

Внешний вид кнопок, размещенных в нижней части окна, зависит от типа диалога и набора вариантов. Вызывая методы `showMessageDialog()` или `showInputDialog()`, вы ограничиваетесь стандартным набором кнопок (OK либо OK и Cancel). Вызывая метод `showConfirmDialog()`, можно выбрать один из четырех наборов вариантов.

```
DEFAULT_OPTION
YES_NO_OPTION
YES_NO_CANCEL_OPTION
OK_CANCEL_OPTION
```

С помощью метода `showOptionDialog()` можно указать произвольный набор вариантов, задав массив объектов, соответствующих каждому из них. Элементы этого массива отображаются на экране следующим образом.

`String`	Создает кнопку, меткой которой является заданная строка.
`Icon`	Создает кнопку, меткой которой является заданная пиктограмма.
`Component`	Отображает компонент.
Любой другой объект	Применяет метод `toString` и создает кнопку, меткой которой является полученная в результате строка.

Статические методы, предназначенные для создания диалоговых окон, возвращают следующие значения.

`showMessageDialog`	Возвращаемое значение отсутствует.
`showConfirmDialog`	Целое число, соответствующее выбранному варианту.
`showOptionDialog`	Целое число, соответствующее выбранному варианту.
`showInputDialog`	Строка, введенная или выбранная пользователем.

Методы showConfirmDialog() и showOptionDialog() возвращают целое число, обозначающее кнопку, на которой щелкнул пользователь. В диалоговом окне для выбора вариантов это число является порядковым номером. Если вместо выбора варианта пользователь закрыл диалоговое окно, возвращается значение CLOSED_OPTION,. Константы, используемые в качестве возвращаемых значений, приведены ниже.

```
OK_OPTION
CANCEL_OPTION
YES_OPTION
NO_OPTION
CLOSED_OPTION
```

Несмотря на обилие мнемонических обозначений, создать диалоговое окно достаточно просто.

1. Выберите тип диалогового окна (сообщение, подтверждение, выбор или ввод).
2. Выберите пиктограмму (ошибка, информация, предупреждение, вопрос, своя собственная или никакой).
3. Выберите сообщение (строка, пиктограмма, пользовательский компонент или массив компонентов).
4. Если вы выбрали диалоговое окно для подтверждения выбора, задайте тип вариантов (по умолчанию, Yes/No, No/Cancel или OK/Cancel).
5. Если вы выбрали диалоговое окно для выбора вариантов, задайте варианты (строки, пиктограммы или собственные компоненты), а также вариант, выбираемый по умолчанию.
6. Если вы выбрали диалоговое окно для ввода данных, выберите поле редактирования или раскрывающийся список.
7. Найдите подходящий метод в классе JOptionPane.

Предположим, что вы хотите отобразить на экране диалоговое окно, показанное на рис. 9.46. В этом окне выводится сообщение, а пользователю предлагается подтвердить его или отклонить. Следовательно, это – диалоговое окно для подтверждения выбора. Пиктограмма, отображаемая в этом окне, относится к предупреждениям. Сообщение является строкой. Набор вариантов задается константой OK_CANCEL_OPTION. Итак, для создания такого диалогового окна выполните следующее:

```
int selection = JOptionPane.showConfirmDialog(parent,
   "Message", "Title",
   JOptionPane.OK_CANCEL_OPTION,
   JOprionPane.WARNING_MESSAGE);
if (selection == JOptionPane.OK_OPTION) ...
```

Строка сообщения может содержать символ перевода строки ('\n'). В этом случае сообщение выводится в виде нескольких строк.

Программа, приведенная в листинге 9.18, позволяет с помощью окна, показанного на рис. 9.47, сделать требуемые установки, генерирует с учетом установок диалоговое окно и отображает его.

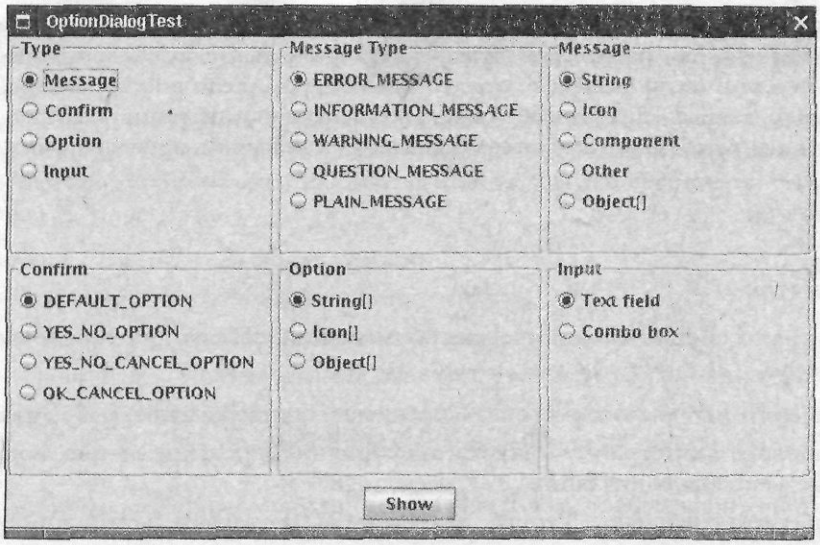


Рис. 9.47. Окно программы `OptionDialogTest`

Листинг 9.18. Содержимое файла `OptionDialogTest.java`

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;
import javax.swing.border.*;

public class OptionDialogTest
{
   public static void main(String[] args)
   {
      OptionDialogFrame frame = new OptionDialogFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
    Панель с переключателями.
*/
class ButtonPanel extends JPanel
{
   /**
      Конструктор панели.
      @param title Заголовок на рамке
      @param options Массив меток переключателей
   */
   public ButtonPanel(String title, String[] options)
   {
      setBorder(BorderFactory.createTitledBorder
```

```
            (BorderFactory.createEtchedBorder(), title));
        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
        group = new ButtonGroup();

        // Для каждого заголовка создается переключатель.
        for (int i = 0; i < options.length; i++)
        {
            JRadioButton b = new JRadioButton(options[i]);
            b.setActionCommand(options[i]);
            add(b);
            group.add(b);
            b.setSelected(i == 0);
        }
    }

    /**
        Определение текущего выбора.
        @return Метка выбранного переключателя.
    */
    public String getSelection()
    {
        return group.getSelection().getActionCommand();
    }

    private ButtonGroup group;
}

/**
    Фрейм, позволяющий выбирать характеристики диалогового окна.
*/
class OptionDialogFrame extends JFrame
{
    public OptionDialogFrame()
    {
        setTitle("OptionDialogTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        JPanel gridPanel = new JPanel();
        gridPanel.setLayout(new GridLayout(2, 3));

        typePanel = new ButtonPanel("Type",
            new String[]
            {
                "Message",
                "Confirm",
                "Option",
                "Input"
            });

        messageTypePanel = new ButtonPanel("Message Type",
            new String[]
            {
                "ERROR_MESSAGE",
                "INFORMATION_MESSAGE",
                "WARNING_MESSAGE",
                "QUESTION_MESSAGE",
```

```
            "PLAIN_MESSAGE"
         });

      messagePanel = new ButtonPanel("Message",
         new String[]
         {
            "String",
            "Icon",
            "Component",
            "Other",
            "Object[]"
         });

      optionTypePanel = new ButtonPanel("Confirm",
         new String[]
         {
            "DEFAULT_OPTION",
            "YES_NO_OPTION",
            "YES_NO_CANCEL_OPTION",
            "OK_CANCEL_OPTION"
         });

      optionsPanel = new ButtonPanel("Option",
         new String[]
         {
            "String[]",
            "Icon[]",
            "Object[]"
         });

      inputPanel = new ButtonPanel("Input",
         new String[]
         {
            "Text field",
            "Combo box"
         });

      gridPanel.add(typePanel);
      gridPanel.add(messageTypePanel);
      gridPanel.add(messagePanel);
      gridPanel.add(optionTypePanel);
      gridPanel.add(optionsPanel);
      gridPanel.add(inputPanel);

      // Добавление панели с кнопкой Show

      JPanel showPanel = new JPanel();
      JButton showButton = new JButton("Show");
      showButton.addActionListener(new ShowAction());
      showPanel.add(showButton);

      add(gridPanel, BorderLayout.CENTER);
      add(showPanel, BorderLayout.SOUTH);
   }

   /**
```

```
   Получение выбранного сообщения.
   @return Строка, пиктограмма, компонент или массив объектов
   в зависимости от выбора на панели Message
*/
public Object getMessage()
{
   String s = messagePanel.getSelection();
   if (s.equals("String"))
      return messageString;
   else if (s.equals("Icon"))
      return messageIcon;
   else if (s.equals("Component"))
      return messageComponent;
   else if (s.equals("Object[]"))
      return new Object[]
      {
         messageString,
         messageIcon,
         messageComponent,
         messageObject
      };
   else if (s.equals("Other"))
      return messageObject;
   else return null;
}

/**
   Получение выбранной опции.
   @return Массив строк, пиктограмм или объектов,
   в зависимости от выбора на панели Option
*/
public Object[] getOptions()
{
   String s = optionsPanel.getSelection();
   if (s.equals("String[]"))
      return new String[] { "Yellow", "Blue", "Red" };
   else if (s.equals("Icon[]"))
      return new Icon[]
      {
         new ImageIcon("yellow-ball.gif"),
         new ImageIcon("blue-ball.gif"),
         new ImageIcon("red-ball.gif")
      };
   else if (s.equals("Object[]"))
      return new Object[]
      {
         messageString,
         messageIcon,
         messageComponent,
         messageObject
      };
   else
      return null;
}
```

```
/**
   Получение типа сообщения или набора вариантов.
   @param panel Панель Message Type или Confirm
   @return Константа XXX_MESSAGE или XXX_OPTION,
   выбранная в классе JOptionPane
*/
public int getType(ButtonPanel panel)
{
   String s = panel.getSelection();
   try
   {
      return JOptionPane.class.getField(s).getInt(null);
   }
   catch(Exception e)
   {
      return -1;
   }
}

/**
   Обработчик событий для кнопки Show. Отображает
   диалоговое окно, тип которого зависит от выбора
   на панели Type.
*/
private class ShowAction implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      if (typePanel.getSelection().equals("Confirm"))
         JOptionPane.showConfirmDialog(
            OptionDialogFrame.this,
            getMessage(),
            "Title",
            getType(optionTypePanel),
            getType(messageTypePanel));
      else if (typePanel.getSelection().equals("Input"))
      {
         if (inputPanel.getSelection().equals("Text field"))
            JOptionPane.showInputDialog(
               OptionDialogFrame.this,
               getMessage(),
               "Title",
               getType(messageTypePanel));
         else
            JOptionPane.showInputDialog(
               OptionDialogFrame.this,
               getMessage(),
               "Title",
               getType(messageTypePanel),
               null,
               new String[] { "Yellow", "Blue", "Red" },
               "Blue");
      }
      else if (typePanel.getSelection().equals("Message"))
         JOptionPane.showMessageDialog(
            OptionDialogFrame.this,
```

```
                getMessage(),
                "Title",
                getType(messageTypePanel));
          else if (typePanel.getSelection().equals("Option"))
             JOptionPane.showOptionDialog(
                OptionDialogFrame.this,
                getMessage(),
                "Title",
                getType(optionTypePanel),
                getType(messageTypePanel),
                null,
                getOptions(),
                getOptions()[0]);
       }
    }

    public static final int DEFAULT_WIDTH = 600;
    public static final int DEFAULT_HEIGHT = 400;

    private ButtonPanel typePanel;
    private ButtonPanel messagePanel;
    private ButtonPanel messageTypePanel;
    private ButtonPanel optionTypePanel;
    private ButtonPanel optionsPanel;
    private ButtonPanel inputPanel;

    private String messageString = "Message";
    private Icon messageIcon = new ImageIcon("blue-ball.gif");
    private Object messageObject = new Date();
    private Component messageComponent = new SamplePanel();
 }

/**
   Панель с графическим изображением.
*/

class SamplePanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;
      Rectangle2D rect = new Rectangle2D.Double(0, 0,
         getWidth() - 1, getHeight() - 1);
      g2.setPaint(Color.YELLOW);
      g2.fill(rect);
      g2.setPaint(Color.BLUE);
      g2.draw(rect);
   }

   public Dimension getMinimumSize()
   {
      return new Dimension(10, 10);
   }
}
```

API javax.swing.JOptionPane 1.2

- `static void showMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)`
- `static void showMessageDialog(Component parent, Object message, String title, int messageType)`
- `static void showMessageDialog(Component parent, Object message)`
- `static void showInternalMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)`
- `static void showInternalMessageDialog(Component parent, Object message, String title, int messageType)`
- `static void showInternalMessageDialog(Component parent, Object message)`

Выводит на экран обычное диалоговое окно или внутреннее диалоговое окно с сообщением. (Внутреннее диалоговое окно целиком содержится внутри фрейма.)

Параметры		
	`parent`	Родительский компонент (значение этого параметра может быть равно `null`)
	`message`	Сообщение, которое выводится в диалоговом окне (может быть строкой, пиктограммой, компонентом или массивом компонентов)
	`title`	Строка заголовка диалогового окна
	`messageType`	Одна из констант: `ERROR_MESSAGE`, *`INFORMATION_MESSAGE`*, *`WARNING_MESSAGE`*, `QUESTION_MESSAGE` или `PLAIN_MESSAGE`
	`icon`	Пиктограмма, которая выводится вместо стандартной

- `static int showConfirmDialog(Component parent, Object message, String title, int optionType, Icon icon)`
- `static int showConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)`
- `static int showConfirmDialog(Component parent, Object message, String title, int optionType)`
- `static int showConfirmDialog(Component parent, Object message)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, Icon icon)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType)`
- `static int showInternalConfirmDialog(Component parent, Object message)`

Выводит на экран диалоговое или внутреннее диалоговое окно для подтверждения выбора. (Внутреннее диалоговое окно целиком отображается внутри

собственного фрейма.) Возвращает вариант, выбранный пользователем (одну из констант: OK_OPTION, YES_OPTION или NO_OPTION), или CLOSED_OPTION, если пользователь закрыл диалоговое окно.

Параметры		
	parent	Родительский компонент (значение этого параметра может быть равно null)
	message	Сообщение, которое выводится в диалоговом окне (может быть строкой, пиктограммой, компонентом или массивом компонентов)
	title	Строка заголовка диалогового окна
	messageType	Одна из констант: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE или PLAIN_MESSAGE
	optionType	Одна из констант: DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION или OK_CANCEL_OPTION
	icon	Пиктограмма, которая выводится вместо стандартной

- `static int showOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)`
- `static int showInternalOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)`

Выводит на экран диалоговое окно или внутреннее диалоговое окно для выбора варианта. (Внутреннее диалоговое окно целиком отображается внутри собственного фрейма.) Возвращает номер варианта, выбранного пользователем, или константу CLOSED_OPTION, если пользователь закрыл диалоговое окно.

Параметры		
	parent	Родительский компонент (значение этого параметра может быть равно null)
	message	Сообщение, которое выводится в диалоговом окне (может быть строкой, пиктограммой, компонентом или массивом компонентов)
	title	Строка заголовка диалогового окна
	messageType	Одна из констант: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE или PLAIN_MESSAGE
	optionType	Одна из констант: DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION или OK_CANCEL_OPTION
	icon	Пиктограмма, которая выводится вместо стандартной

	`options`	Массив вариантов, элементами которого могут быть строки, пиктограммы или компоненты
	`default`	Вариант, предлагаемый по умолчанию

- `static Object showInputDialog(Component parent, Object message, String title, int messageType, Icon icon, Object[] values, Object default)`
- `static String showInputDialog(Component parent, Object message, String title, int messageType)`
- `static String showInputDialog(Component parent, Object message)`
- `static String showInputDialog(Object message)`
- `static String showInputDialog(Component parent, Object message, Object default)` **1.4**
- `static String showInputDialog(Object message, Object default)` **1.4**
- `static Object showInternalInputDialog(Component parent, Object message, String title, int messageType, Icon icon, Object[] values, Object default)`
- `static String showInternalInputDialog(Component parent, Object message, String title, int messageType)`
- `static Object showInternalInputDialog(Component parent, Object message)`

Выводит на экран диалоговое окно или внутреннее диалоговое окно для ввода данных. (Внутреннее диалоговое окно целиком отображается внутри собственного фрейма.) Возвращает строку, введенную пользователем, или значение `null`, если пользователь закрыл окно.

Параметры	`parent`	Родительский компонент (значение этого параметра может быть равно `null`)
	`message`	Сообщение, которое выводится в диалоговом окне (может быть строкой, пиктограммой, компонентом или массивом компонентов)
	`title`	Строка заголовка диалогового окна
	`messageType`	Одна из констант: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE` или `PLAIN_MESSAGE`
	`optionType`	Одна из констант: `DEFAULT_OPTION`, `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION` или `OK_CANCEL_OPTION`
	`icon`	Пиктограмма, которая выводится вместо стандартной
	`values`	Массив значений, предлагаемых в раскрывающемся списке
	`default`	Значение, предлагаемое пользователю по умолчанию

Создание диалоговых окон

Как было сказано ранее, для использования предопределенных диалоговых окон применяется класс `JOptionPane`. Теперь мы опишем, как создать свое собственное диалоговое окно.

На рис. 9.48 показано типичное модальное диалоговое окно, содержащее сообщение. Подобное окно отображается на экране после того, как пользователь выбрал пункт меню About.

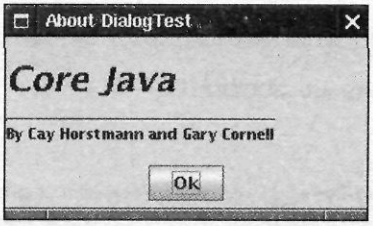


Рис. 9.48. Диалоговое окно About

Для того чтобы реализовать такое окно, нужно создать подкласс класса `JDialog`. По существу, это тот же процесс, которым мы воспользовались для создания обычного окна, расширяя класс `JFrame`. Точнее говоря, нужно сделать следующее.

1. В конструкторе вашего диалогового окна вызовите конструктор суперкласса `JDialog`. Ему нужно указать *фрейм-владелец* (окно фрейма, в котором будет появляться диалоговое окно), заголовок окна и логическую переменную, означающую, является ли данное диалоговое окно модальным или нет.

   Фрейм-владелец указывается для того, чтобы диалоговое окно могло отображаться поверх него. Если вам все равно, где будет отображаться ваше окно, задайте этот параметр равным `null`. Обычно системы оконного интерфейса требуют, чтобы с каждым диалоговым окном был связан свой фрейм. По этой причине библиотека Swing создает невидимый фрейм, предназначенный для совместного пользования, который становится фреймом-владельцем всех окон, у которых данный параметр равен `null`.
2. Добавьте в диалоговое окно компоненты пользовательского интерфейса.
3. Добавьте обработчики событий.
4. Задайте размер диалогового окна.

Ниже приведен фрагмент программы, создающий диалоговое окно.

```
public AboutDialog extends JDialog
{
   public AboutDialog(JFrame owner)
   {
      super(owner, "About DialogTest", true);
      add(new JLabel(
         "<html><h1><i>Core Java</i></h1><hr>"+
         "By Cay Horstmann and Gary Cornell</html>"),
          BorderLayout.CENTER);
```

```
      JPanel panel = new JPanel();
      JButton ok = new JButton("OK");

      ok.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               setVisible(false);
            }
         });

      panel.add(ok);
      add(panel, BorderLayout.SOUTH);

      setSize(250, 150);
   }
}
```

Как видим, конструктор добавляет элементы пользовательского интерфейса, в данном случае – метки и кнопку. Кроме того, он добавляет обработчик кнопки и задает размер окна.

Для того чтобы отобразить диалоговое окно на экране, нужно создать новый объект `JDialog` и вызвать метод `setVisible()`.

```
JDialog dialog = new AboutDialog(this);
dialog.setVisible(true);
```

Фактически в программе, рассматриваемой здесь в качестве примера, диалоговое окно создается только единожды, а затем мы используем его повторно каждый раз, когда пользователь щелкает на кнопке About.

```
if (dialog == null) // Первый раз.
   dialog == new AboutDialog(this);
dialog.setVisible(true);
```

Когда пользователь щелкает на кнопке OK, диалоговое окно должно закрываться. Эта реакция определяется в обработчике событий, связанных с кнопкой OK.

```
ok.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent evt)
      {
         setVisible(false);
      }
   });
```

Когда пользователь закрывает окно диалога, щелкая на кнопке Close, оно исчезает с экрана. Как и в классе `JFrame`, программист может изменить эту реакцию с помощью метода `setDefaultCloseOperation()`.

Программа, демонстрирующая использование диалогового окна About, приведена в листинге 9.19.

Листинг 9.19. Содержимое файла DialogTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DialogTest
{
   public static void main(String[] args)
   {
      DialogFrame frame = new DialogFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм содержит меню. При выборе пункта File->About
   отображается диалоговое окно.
*/
class DialogFrame extends JFrame
{
   public DialogFrame()
   {
      setTitle("DialogTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Создание меню File.

      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);
      JMenu fileMenu = new JMenu("File");
      menuBar.add(fileMenu);

      // Добавление пунктов меню About и Exit.

      // При выборе пункта About отображается
      // диалоговое окно About.

      JMenuItem aboutItem = new JMenuItem("About");
      aboutItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               if (dialog == null) // first time
                  dialog = new AboutDialog(DialogFrame.this);
               dialog.setVisible(true); // pop up dialog
            }
         });
      fileMenu.add(aboutItem);

      // При активизации пункта Exit программа завершается.

      JMenuItem exitItem = new JMenuItem("Exit");
      exitItem.addActionListener(new
```

```
            ActionListener()
            {
               public void actionPerformed(ActionEvent event)
               {
                  System.exit(0);
               }
            });
         fileMenu.add(exitItem);
      }

      public static final int DEFAULT_WIDTH = 300;
      public static final int DEFAULT_HEIGHT = 200;

      private AboutDialog dialog;
   }

   /**
      Модальное диалоговое окно отображает сообщение и
      ожидает щелчка на кнопке OK.
   */
   class AboutDialog extends JDialog
   {
      public AboutDialog(JFrame owner)
      {
         super(owner, "About DialogTest", true);

         // Метка, содержащая HTML-форматирование, выравнивается
         // по центру.

         add(new JLabel(
            "<html><h1><i>Core Java</i></h1><hr>"+"
            By Cay Horstmann and  Gary Cornell</html>"),
            BorderLayout.CENTER);

         // При активизации кнопки OK диалоговое окно закрывается.

         JButton ok = new JButton("OK");
         ok.addActionListener(new
            ActionListener()
            {
               public void actionPerformed(ActionEvent event)
               {
                  setVisible(false);
               }
            });

         // Кнопка OK помещается в нижнюю часть окна.

         JPanel panel = new JPanel();
         panel.add(ok);
         add(panel, BorderLayout.SOUTH);

         setSize(250, 150);
      }
   }
```

javax.swing.JDialog 1.2

- `public JDialog(JFrame parent, String title, boolean modal)`

  Создает диалоговое окно, которое является невидимым, пока не будет принудительно выведено на экран.

Параметры	`parent`	Родительский компонент (значение этого параметра может быть равно `null`)
	`title`	Строка заголовка диалогового окна
	`modal`	Если значение этого параметра равно `true`, диалоговое окно является модальным (блокирует доступ к остальным окнам)

Обмен данными

Чаще всего диалоговые окна создаются для того, чтобы дать возможность пользователю вводить данные в программу. Выше было показано, насколько легко создавать объекты класса `JDialog`: задайте исходные данные, а затем вызывайте метод `setVisible(true)`, который выведет окно на экран. Теперь посмотрим, как можно вводить данные с помощью диалогового окна.

Обратите внимание на окно, показанное на рис. 9.49. Его можно использовать для получения имени пользователя и пароля.

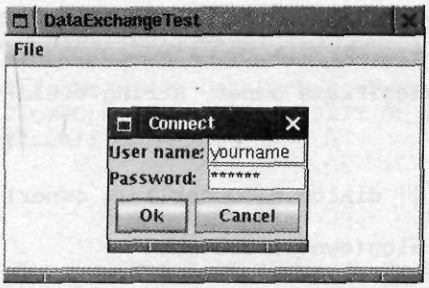


Рис. 9.49. Диалоговое окно для ввода пароля

В диалоговом окне должны быть предусмотрены методы, генерирующие данные по умолчанию. Например, в классе `PasswordChooser` есть метод `setUser`.

```
public void setUser(User u)
{
   userName.setText(u.getName());
}
```

Установив при необходимости значения по умолчанию, окно выводится на экран. Для этого вызывается метод `setVisible(true)`.

Пользователь должен ввести информацию и щелкнуть на кнопке OK или Cancel. Обработчики событий, связанных с обеими кнопками, вызывают метод `setVisible(false)`. Этот вызов выполняет действия, обратные `setVisible(true)`. Кроме того, пользователь может прекратить обмен, закрыв окно. Если обработчик событий, связанных с диалоговым окном, не установлен, то выполняются обычные операции по закрытию окон: диалоговое окно становится невидимым.

Обратите внимание на то, что вызов метода `setVisible(true)` блокирует выполнение программы до тех пор, пока пользователь не выполнит действие, приводящее к закрытию окна. Такой подход существенно упрощает создание диалоговых окон и их использование.

В большинстве случаев разработчику программы необходима информация о том, подтвердил ли пользователь введенную им информацию. В программе, рассматриваемой в качестве примера, используется следующий подход. Перед обращением к диалоговому окну устанавливается значение `false` переменной `ok`. Значение `true` присваивается этой переменной только в обработчике, связанном с кнопкой OK. В этом случае введенные пользователем данные могут быть использованы программой.

Обмен данными с немодальным диалоговым окном является непростой задачей. При отображении такого окна вызов `setVisible(true)` не блокирует выполнение программы. Если пользователь выполнит некоторые установки в немодальном диалоговом окне и щелкнет на кнопке OK, это окно должно оповестить некоторый обработчик.

В рассматриваемой здесь программе есть и другие важные особенности. При создании объекта класса `JDialog` должен быть указан фрейм-владелец. Однако зачастую приходится отображать на экране одно и то же диалоговое окно с разными фреймами. Гораздо удобнее, если фрейм-владелец задается, *когда диалоговое окно готово к выводу на экран*, а не во время создания объекта класса `PasswordChooser`.

Этого можно добиться, если класс `PasswordChooser` будет расширять класс `JPanel`, а не класс `JDialog`. Попробуем создать объект `JDialog` при выполнении метода `showDialog()`.

```
public boolean showDialog(Frame owner, String title)
{
   ok = false;

   if (dialog == null || dialog.getOwner() != owner)
   {
      dialog = new JDialog(owner, true);
      dialog.add(this);
      dialog.pack();
   }

   dialog.setTitle(title);
   dialog.setVisible(true);
   return ok;
}
```

Отметим, что для большей безопасности значение ссылки `owner` должно быть равно `null`.

Иногда фрейм-владелец бывает недоступен. Однако его можно вычислить с помощью объекта `parent`.

```
Frame owner;
if (parent instanceof Frame)
   owner = (Frame) parent;
else
   owner = (Frame) SwingUtilities.getAncestorOfClass
      (Frame.class, parent);
```

Этот подход применяется в нашей программе. Класс JOptionPane также использует этот механизм.

Многие диалоговые окна имеют кнопку, заданную по умолчанию, которая выбирается автоматически, если пользователь нажимает клавишу ввода (в большинстве стилей этой клавишей является <Enter>). Кнопка, заданная по умолчанию, выделяется среди остальных, чаще всего контуром.

Для установки кнопки по умолчанию на *корневой панели* (root pane) диалогового окна нужно вызвать следующий метод:

```
dialog.getRootPane().setDefaultButton(okButton);
```

Если вы собираетесь разместить интерфейсные элементы на панели, будьте осторожны: устанавливайте кнопку по умолчанию только после оформления панели в виде диалогового окна. Панель сама по себе не имеет корневой панели.

В листинге 9.20 приведен полный текст программы, иллюстрирующей обмен данными с диалоговым окном.

Листинг 9.20. Содержимое файла DataExchangeTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DataExchangeTest
{
   public static void main(String[] args)
   {
      DataExchangeFrame frame = new DataExchangeFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с меню. При выборе пункта File->Connect отображается
   диалоговое окно для ввода пароля.
*/
class DataExchangeFrame extends JFrame
{
   public DataExchangeFrame()
   {
      setTitle("DataExchangeTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Создание меню File.

      JMenuBar mbar = new JMenuBar();
      setJMenuBar(mbar);
      JMenu fileMenu = new JMenu("File");
      mbar.add(fileMenu);

      // Создание пунктов Connect и Exit.

      JMenuItem connectItem = new JMenuItem("Connect");
```

```
      connectItem.addActionListener(new ConnectAction());
      fileMenu.add(connectItem);

      // При активизации пункта меню Exit программа завершается.

      JMenuItem exitItem = new JMenuItem("Exit");
      exitItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });
      fileMenu.add(exitItem);

      textArea = new JTextArea();
      add(new JScrollPane(textArea), BorderLayout.CENTER);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   private PasswordChooser dialog = null;
   private JTextArea textArea;

   /**
      Обработчик, связанный с кнопкой Connect,
      отображает диалоговое окно для ввода пароля.
   */

   private class ConnectAction implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         // При первом обращении создается диалоговое окно.

         if (dialog == null)
            dialog = new PasswordChooser();

         // Установка значений по умолчанию.
         dialog.setUser(new User("yourname", null));

         // Вывод диалогового окна.
         if (dialog.showDialog(DataExchangeFrame.this, "Connect"))
         {
            // Если пользователь подтвердил введенные
            // данные, они используются в программе.
            User u = dialog.getUser();
            textArea.append(
               "user name = " + u.getName()
               + ", password = " + (new String(u.getPassword()))
               + "\n");
         }
      }
   }
```

```
}

/**
   Окно для ввода пароля, отображаемое в
   диалоговом окне
*/
class PasswordChooser extends JPanel
{
   public PasswordChooser()
   {
      setLayout(new BorderLayout());

      // Создание панели с полями для ввода
      // имени пользователя и пароля.

      JPanel panel = new JPanel();
      panel.setLayout(new GridLayout(2, 2));
      panel.add(new JLabel("User name:"));
      panel.add(username = new JTextField(""));
      panel.add(new JLabel("Password:"));
      panel.add(password = new JPasswordField(""));
      add(panel, BorderLayout.CENTER);

      // Создание кнопок OK и Cancel

      okButton = new JButton("OK");
      okButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               ok = true;
               dialog.setVisible(false);
            }
         });

      JButton cancelButton = new JButton("Cancel");
      cancelButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               dialog.setVisible(false);
            }
         });

      // Включение кнопок в нижнюю часть окна.

      JPanel buttonPanel = new JPanel();
      buttonPanel.add(okButton);
      buttonPanel.add(cancelButton);
      add(buttonPanel, BorderLayout.SOUTH);
   }

   /**
      Установка значений по умолчанию.
```

```
      @param u Информация о пользователе, принятая по умолчанию
   */
   public void setUser(User u)
   {
      username.setText(u.getName());
   }

   /**
      Получение введенной информации.
      @return Объект User, состояние которого отражает
         введенные данные
   */
   public User getUser()
   {
      return new User(username.getText(), password.getPassword());
   }

   /**
      Отображение панели для ввода данных в диалоговом окне.
      @param parent Компонент фрейма-владельца или null
      @param title Заголовок диалогового окна
   */
   public boolean showDialog(Component parent, String title)
   {
      ok = false;

      // Обнаружение фрейма-владельца.

      Frame owner = null;
      if (parent instanceof Frame)
         owner = (Frame) parent;
      else
         owner = (Frame)SwingUtilities.getAncestorOfClass
            (Frame.class, parent);

      // При первом обращении или при изменении фрейма-владельца
      // создается новое диалоговое окно.

      if (dialog == null || dialog.getOwner() != owner)
      {
         dialog = new JDialog(owner, true);
         dialog.add(this);
         dialog.getRootPane().setDefaultButton(okButton);
         dialog.pack();
      }

      // Установка заголовка и отображение диалогового окна.

      dialog.setTitle(title);
      dialog.setVisible(true);
      return ok;
   }

   private JTextField username;
   private JPasswordField password;
   private JButton okButton;
```

```
   private boolean ok;
   private JDialog dialog;
}

/**
   Для пользователя определены имя и пароль. Из соображений
   безопасности пароль содержится не в объекте String, а в
   массиве char[].
*/
class User
{
   public User(String aName, char[] aPassword)
   {
      name = aName;
      password = aPassword;
   }

   public String getName() { return name; }
   public char[] getPassword() { return password; }

   public void setName(String aName) { name = aName; }
   public void setPassword(char[] aPassword)
      { password = aPassword; }

   private String name;
   private char[] password;
}
```

javax.swing.SwingUtilities 1.2

- `Container getAncestorOfClass(Class c, Component comp)`

  Возвращает наиболее глубоко вложенный родительский контейнер указанного компонента, принадлежащего заданному классу или одному из его подклассов.

javax.swing.JComponent 1.2

- `JRootPane getRootPane()`

  Определяет корневую панель, содержащую заданный компонент. Если компонент не имеет предка с корневой панелью, возвращает ссылку `null`.

javax.swing.JRootPane 1.2

- `void setDefaultButton(JButton button)`

  Устанавливает на корневой панели кнопку по умолчанию. Чтобы заблокировать кнопку по умолчанию, этот метод вызывается с параметром `null`.

javax.swing.JButton 1.2

- `boolean isDefaultButton()`

  Возвращает значение `true`, если данная кнопка является кнопкой по умолчанию на корневой панели.

Диалоговые окна для выбора файлов

Во многих приложениях нужно открывать и сохранять файлы. Хорошее диалоговое окно, позволяющее пользователю свободно перемещаться по файловой системе, создать довольно трудно. Да это и не нужно – ведь вы не собираетесь изобретать колесо! В библиотеке Swing есть класс `JFileChooser`, позволяющий отображать на экране диалоговое окно для выбора файлов, которое удовлетворяет потребностям большинства приложений. Это диалоговое окно всегда является модальным. Обратите внимание: класс `JFileChooser` не расширяет класс `JDialog`. Вместо вызова метода `setVisible(true)` для отображения диалогового окна, предназначенного для открытия файлов, вызывается метод `showOpenDialog()`, а для сохранения файлов – метод `showSaveDialog()`. Кнопка, подтверждающая выбор файла, автоматически помечается как Open или Save. С помощью метода `showDialog()` можно задать свою собственную метку кнопки. На рис. 9.50 показан пример диалогового окна для выбора файлов.

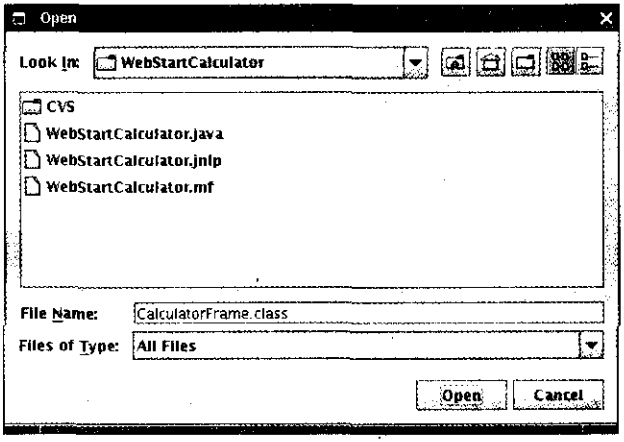


Рис. 9.50. Диалоговое окно для выбора файлов

Чтобы отобразить на экране диалоговое окно для выбора файлов и определить, какой выбор сделал пользователь, выполните следующие действия.

1. Создайте объект класса `JFileChooser`. В отличие от конструктора класса `JDialog`, родительский компонент указывать не нужно. Такое решение позволяет повторно использовать диалоговое окно для выбора файлов при работе с различными фреймами.

```
JFileChooser chooser = new JFileChooser();
```

Повторное использование диалогового окна для выбора файлов — прекрасная идея, поскольку конструктор класса `JFileChooser` иногда работает довольно медленно. Особенно это заметно в среде Windows, при работе с распределенной файловой системой.

2. Укажите каталог, вызвав метод `setCurrentDirectory()`. Например, чтобы задать текущий каталог, нужно использовать следующее выражение:

```
chooser.setCurrentDirectory(new File("."));
```

   При этом необходимо задать объект `File`. Детально класс `File` описывается в главе 12, а пока вам достаточно знать, что в этом классе есть конструктор `File(String filename)`, преобразовывающий строку объекта `File`.

3. Если нужно предложить пользователю имя файла, выбираемого по умолчанию, задайте его с помощью метода `setSelectedFile()`.

```
chooser.setSelectedFile(new File(filename));
```

4. Чтобы дать пользователю возможность выбрать одновременно несколько файлов, вызовите метод `setMultiSelectionEnabled()`. Разумеется, делать это совершенно не обязательно, поскольку подобная возможность нужна далеко не всегда.

```
chooser.setMultiSelectionEnabled(true);
```

5. Для того чтобы ограничить выбор пользователя файлами, имеющими определенное расширение (например, расширение `.gif`), нужно задать *фильтр файлов* (file filter). Этот вопрос мы обсудим немного позже.

6. По умолчанию пользователь может выбирать в диалоговом окне `JFileChooser` только файлы. Если вы хотите предоставить пользователю возможность выбирать целые каталоги, используйте метод `setFileSelectionMode()` и вызывайте его со следующими параметрами:

   `JFileChooser.FILES_ONLY` (*по умолчанию*)
   `JFileChooser.DIRECTORIES_ONLY` или
   `JFileChooser.FILES_AND_DIRECTORIES`.

7. Диалоговое окно выводится на экран с помощью методов `showOpenDialog()` или `showSaveDialog()`. При вызове этих методов нужно указывать родительский компонент.

```
int result = chooser.showOpenDialog(parent);
int result = chooser.showSaveDialog(parent);
```

   Единственное отличие между этими вызовами заключается в метке "кнопки выбора", т.е. кнопки, на которой пользователь щелкает, выбирая файл. Можно также вызвать метод `showDialog` и явно задать текст на кнопке.

```
int result = chooser.showDialog(parent, "Select");
```

   Эти вызовы возвращают управление только тогда, когда пользователь сделал выбор или прекратил диалог, закрыв окно. Возвращаемое значение может быть одной из следующих констант: `JFileChooser.APPROVE_OPTION`, `JFileChooser.CANCEL_OPTION`, или `JFileChooser.ERROR_OPTION`.

8. Выбранный файл можно получить с помощью методов `getSelectedFile()` *или* `getSelectedFiles()`. *Эти методы возвращают либо один объект класса* `File`, либо массив таких объектов. Если пользователю нужно знать лишь имя файла, вызывайте метод `getPath()`.

```
String filename = chooser.getSelectedFile().getPath();
```

Большинство этих действий довольно просты. Основная трудность связана с использованием диалогового окна для выбора файлов, ограничивающего выбор пользователя. Предположим, что пользователь должен выбирать только графические файлы в формате GIF. Следовательно, в диалоговом окне должны отображаться только файлы, имеющие расширение `.gif`. Кроме того, пользователю нужно подсказать, что это за файлы. Ситуация может оказаться еще сложнее. Графические файлы в формате JPEG могут иметь расширения `.jpg` или `.jpeg`. Для преодоления этих трудностей разработчики предложили элегантный механизм: чтобы ограничить спектр отображаемых файлов, нужно создать подкласс абстрактного класса `javax.swing.filechooser.FileFilter`. Диалоговое окно передает фильтру каждый файл и отображает только файлы, прошедшие его.

На момент написания этой книги был известен только один такой подкласс: фильтр, задаваемый по умолчанию и пропускающий все файлы.

Создать свой собственный фильтр довольно легко. Доататочно реализовать два метода, которые в классе `FileFilter` объявлены как абстрактные.

```
public boolean accept(File f)
public String getDescription();
```

В пакете `java.io` также существует интерфейс `FileFilter` (совершенно независимый от описанного выше абстрактного класса). В этом интерфейсе объявлен только один метод — `boolean accept(File f)`. Он используется в методе `listFiles()` класса `File`. Совершенно непонятно, почему разработчики библиотеки Swing не стали расширять этот интерфейс, — возможно, библиотека классов языка Java настолько сложна, что даже программисты компании Sun не знают обо всех стандартных классах и интерфейсах.

Если в программу одновременно импортируются пакеты `java.io` и `javax.swing.filterchooser`, нам снова придется разрешать конфликт имен Проще всего вместо пакета `javax.swing.filechooser.*` импортировать класс `javax.swing.filechooser.FileFilter`.

Первый из приведенных выше методов проверяет, соответствует ли файл наложенным ограничениям. Второй метод возвращает описание типа файла, который может отображаться в диалоговом окне. Например, для отбора графических файлов в формате GIF можно использовать следующий фрагмент кода:

```
public class GifFilter extends FileFilter
{
   public boolean accept(File f)
   {
      return f.getName().toLowerCase().endsWith(".gif") ||
         f.isDirectory();
   }
   public String getDescription()
   {
      return "GIF Image";
   }
```

для того чтобы связать экземпляр класса, реализующего интерфейс `FileFilter`, ектом `JFileChooser`, следует применять метод `setFileFilter()`.

```
er.setFileFilter(new GifFilter());
```

В программе, которая будет рассмотрена в качестве примера, предусмотрен класс ExtensionFileFilter, который используется следующим образом:

```
ExtensionFileFilter filter = new ExtensionFileFilter();
filter.addExtension("jpg");
filter.addExtension("jpg");
filter.setDescription("Image files");
```

Реализация класса ExtensionFileFilter представляет собой модификацию класса GifFilter для повышения его функциональных возможностей. Этот класс вы можете использовать и в своих программах.

В JDK аналогичный класс ExampleFileFilter содержится в каталоге demo/jfc/FileChooserDemo.

Инсталлировать несколько фильтров для диалогового окна, предназначенного для выбора файлов, можно с помощью вызовов

```
chooser.addChoosableFileFilter(new GifFilter());
chooser.addChoosableFileFilter(new JpegFilter());
...
```

Пользователь выбирает фильтр с помощью раскрывающегося списка, расположенного внизу окна. По умолчанию в этом списке всегда есть пункт All files. Обычно его присутствие оправдано. Однако если вам потребуется отменить фильтр All files, следует вызвать следующий метод:

```
chooser.setAcceptAllFileFilterUsed(false);
```

Если одно и то же диалоговое окно используется для чтения и записи файлов разных типов, следует вызывать метод

```
chooser.resetChoosableFilters()
```

Это позволит удалить старые фильтры перед добавлением новых.

Отметим также, что в диалоговом окне каждый файл можно сопроводить специальной пиктограммой и кратким описанием. Для этого нужно предусмотреть экземпляр класса, расширяющего класс FileView из пакета javax.swing.filechooser. Это довольно сложно и не всегда оправдано. Обычно внешний вид файла программиста не интересует — настраиваемые стили интерфейса сами решают эту задачу. Однако, если вы захотите сопроводить файлы определенного типа специальными пиктограммами, можно задать свой собственный стиль отображения файлов. Для этого нужно расширить класс FileView и реализовать пять методов.

```
Icon getIcon(File f);
String getName(File f);
String getDescription(File f);
String getTypeDescription(File f);
Boolean isTraversable(File f);
```

Затем следует вызвать метод setFileView(), чтобы связать представление с диалоговым окном.

Диалоговое окно вызывает эти методы для каждого файла или каталога, подлежащего выводу на экран. Если для пиктограммы, имени или описания файла метод возвращает ссылку null, диалоговое окно использует стиль, заданный по умолчанию. Это правильно, поскольку такой подход позволяет применять особый стиль только к отдельным типам файлов.

Для того чтобы решить, открывать ли каталог, выбранный пользователем, диалоговое окно вызывает метод isTraversable. Обратите внимание на то, что этот метод *возвращает объект класса* Boolean, *а не значение типа* boolean! *Это выглядит* довольно загадочно, но удобно — если у вас нет особых требований к стилю, возвращайте нулевую ссылку null. Затем диалоговое окно использует стиль отображения файлов, заданный по умолчанию. Иными словами, метод возвращает объект класса Boolean, чтобы дать вам возможность выбрать одно из трех: да (Boolean.TRUE), нет (Boolean.FALSE) и не важно (null).

Программа, рассматриваемая здесь в качестве примера, содержит простой класс, задающий стиль отображения файлов. Этот класс выводит на экран определенную пиктограмму каждый раз, когда файл проходит фильтр. Мы используем этот класс для отображения на экране палитры пиктограмм для всех графических файлов.

```
class FileIconView extends FileView
{
   public FileIconView(FileFilter aFilter, Icon anIcon)
   {
      filter = aFilter;
      icon = anIcon;
   }

   public Icon getIcon(File f)
   {
      if (!f.isDirectory() && filter.accept(f))
         return icon;
      else return null;
   }

   private FileFilter filter;
   private Icon icon;
}
```

В версии JDK 1.2 необходимо определять в подклассе FileView все пять методов. Если метод не нужен, можно вернуть значение null. В версии JDK 1.3 методы класса FileView больше не являются абстрактными.

Чтобы связать объект класса FileView с диалоговым окном, применяется метод setFileView().

```
chooser.setFileView(new FileIconView(filter,
   new ImageIcon("palette.gif"));));
```

Если файл прошел фильтр, на экран будет выведена соответствующая пиктограмма, а для всех остальных файлов будет использоваться стиль, заданный по умолчанию.

В каталоге demo/jfc/FileChooserDemo пакета JDK можно найти более полезный класс ExampleFileView. Этот класс позволяет связывать пиктограммы и описания с любыми расширениями файлов.

Отметим также, что диалоговое окно для выбора файлов можно снабдить *компонентом доступа* (accessory component). Например, на рис. 9.51 показан подобный компонент, позволяющий отображать наряду со списком файлов содержимое файла, выбранного в текущий момент.

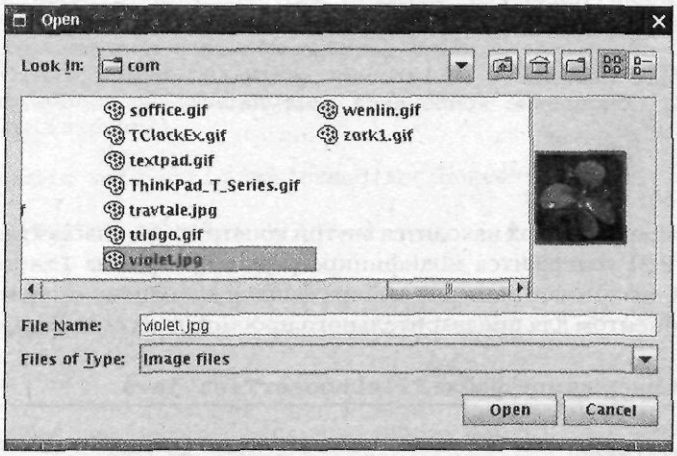


Рис. 9.51. Диалоговое окно для выбора файлов с возможностью предварительного просмотра

Компонентом доступа может быть любой компонент библиотеки Swing. В данном случае мы расширяем класс `JLabel` и используем в качестве пиктограммы уменьшенную копию графического изображения.

```
class ImagePreviewer extends JLabel
{
   public ImagePreviewer(JFileChooser chooser)
   {
      setPreferredSize(new Dimension(100, 100));
      setBorder(BorderFactory.createEtchedBorder());
   }

   public void loadImage(File f)
   {
      ImafeIcon icon = new ImageIcon(f.getPath());
      if(icon.getIconWidth() > getWidth())
         icon = new ImageIcon(icon.getImage().getScaledInstance
               (getWidth(), -1, Image.SCALE_DEFAULT));
      setIcon(icon);
      repaint();
   }
}
```

Осталась лишь одна проблема. Мы хотим обновлять изображение предварительного просмотра всякий раз, когда пользователь выбирает новый файл. Для этого диалоговое окно использует механизм JavaBeans, извещая обработчиков об изменениях своих свойств. Выбранный файл — это свойство, которое можно отслеживать с помощью обработчика `PropertyChangeListener`. Более детально этот механизм обсуждается во втором томе. Ниже показано, каким образом происходит оповещение обработчиков.

```
chooser.addPropertyChangeListener( new
   PropertyChangeListener()
   {
      public void propertyChange(PropertyChangeEvent event)
      {
         if (event.getPropertyName(0) ==
            JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)
         {
            File newfile = (File)event.getNewValue()
            // Обновление компонента доступа.
         }
      }
  });
```

В нашем примере этот код находится внутри конструктора класса `ImagePreviewer`.

В листинге 9.21 содержится модифицированная программа `ImageViewer` из главы 2, в которой диалоговое окно для выбора файлов дополнено стилем представления файлов и компонентом для предварительного просмотра их содержимого.

Листинг 9.21. Содержимое файла `FileChooserTest.java`

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.beans.*;
import java.util.*;
import java.io.*;
import javax.swing.*;
import javax.swing.filechooser.FileFilter;
import javax.swing.filechooser.FileView;

public class FileChooserTest
{
   public static void main(String[] args)
   {
      ImageViewerFrame frame = new ImageViewerFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм, содержащий меню для загрузки графических файлов
   и область для отображения загруженных изображений.
*/
class ImageViewerFrame extends JFrame
{
   public ImageViewerFrame()
   {
      setTitle("FileChooserTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Установка строки меню.
      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);
```

```
      JMenu menu = new JMenu("File");
      menuBar.add(menu);

      JMenuItem openItem = new JMenuItem("Open");
      menu.add(openItem);
      openItem.addActionListener(new FileOpenListener());

      JMenuItem exitItem = new JMenuItem("Exit");
      menu.add(exitItem);
      exitItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });

      // Для вывода изображений используется метка.
      label = new JLabel();
      add(label);

      // Создание диалогового окна для выбора файла.
      chooser = new JFileChooser();

      // Допустимы только графические файлы
      // с расширениями .jpg, .jpeg, .gif
      final ExtensionFileFilter filter = new
         ExtensionFileFilter();
      filter.addExtension("jpg");
      filter.addExtension("jpeg");
      filter.addExtension("gif");
      filter.setDescription("Image files");
      chooser.setFileFilter(filter);

      chooser.setAccessory(new ImagePreviewer(chooser));

      chooser.setFileView(new FileIconView(filter,
         new ImageIcon("palette.gif")));
   }

   /**
      Обработчик событий, связанных с пунктом меню File->Open.
   */
   private class FileOpenListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         chooser.setCurrentDirectory(new File("."));

         // Отображение диалогового окна для выбора файлов.
         int result =
            chooser.showOpenDialog(ImageViewerFrame.this);

         // Если файл является допустимым, он используется
         // в качестве пиктограммы для метки.
```

```
            if(result == JFileChooser.APPROVE_OPTION)
            {
               String name = chooser.getSelectedFile().getPath();
               label.setIcon(new ImageIcon(name));
            }
         }
      }

      public static final int DEFAULT_WIDTH = 300;
      public static final int DEFAULT_HEIGHT = 400;

      private JLabel label;
      private JFileChooser chooser;
}

/**
   Данный фильтр пропускает файлы с расширениями из
   определенного набора.
*/
class ExtensionFileFilter extends FileFilter
{
   /**
      Добавление расширения, распознаваемого фильтром.
      @param extension Расширение (например ".txt" или "txt")
   */
   public void addExtension(String extension)
   {
      if (!extension.startsWith("."))
         extension = "." + extension;
      extensions.add(extension.toLowerCase());
   }

   /**
      Создание описания для файлов, распознаваемых фильтром.
      @param aDescription Описание файлов
   */
   public void setDescription(String aDescription)
   {
      description = aDescription;
   }

   /**
      Возвращает описание файлов, распознаваемых фильтром.
      @return Описание файлов
   */
   public String getDescription()
   {
      return description;
   }

   public boolean accept(File f)
   {
      if (f.isDirectory()) return true;
      String name = f.getName().toLowerCase();

      // Проверка, оканчивается ли имя файла одним из
```

```
      // предусмотренных расширений.
      for (String extension : extensions)
         if (name.endsWith(extension))
            return true;
      return false;
   }

   private String description = "";
   private ArrayList<String> extensions = new ArrayList<String>();
}

/**
   Средства отображения, предоставляющие пиктограмму
   для каждого файла, распознаваемого фильтром
*/
class FileIconView extends FileView
{
   /**
      Конструктор FileIconView.
      @param aFilter Фильтр - все файлы, которые пропускает
        этот фильтр, помечаются специальной пиктограммой.
      @param anIcon Пиктограмма для файлов, распознаваемых
        фильтром
   */
   public FileIconView(FileFilter aFilter, Icon anIcon)
   {
      filter = aFilter;
      icon = anIcon;
   }

   public Icon getIcon(File f)
   {
      if (!f.isDirectory() && filter.accept(f))
         return icon;
      else return null;
   }

   private FileFilter filter;
   private Icon icon;
}

/**
   Компонент доступа, используемый для предварительного
   просмотра изображений.
*/
class ImagePreviewer extends JLabel
{
   /**
      Конструктор ImagePreviewer.
      @param chooser Диалоговое окно для выбора файлов.
         Изменение свойства приводит к изменению
         предварительного просмотра
   */
   public ImagePreviewer(JFileChooser chooser)
   {
      setPreferredSize(new Dimension(100, 100));
```

```
      setBorder(BorderFactory.createEtchedBorder());

      chooser.addPropertyChangeListener(new
         PropertyChangeListener()
         {
            public void propertyChange(PropertyChangeEvent
               event)
            {
               if (event.getPropertyName() ==
                  JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)
               {
                  // Пользователь выбрал новый файл.
                  File f = (File) event.getNewValue();
                  if (f == null) { setIcon(null); return; }

                  // Чтение изображения для использования
                  // в качестве пиктограммы.
                  ImageIcon icon = new ImageIcon(f.getPath());

                  // Если пиктограмма слишком велика,
                  // выполняется масштабирование.
                  if(icon.getIconWidth() > getWidth())
                     icon = new ImageIcon(icon.getImage().
                       getScaledInstance(getWidth(), -1,
                       Image.SCALE_DEFAULT));

                  setIcon(icon);
               }
         }
         });
   }
}
```

API **javax.swing.JFileChooser 1.2**

- `JFileChooser()`

  Создает диалоговое окно для выбора файлов, которое можно использовать *с несколькими фреймами*.

- `void setCurrentDirectory(File dir)`

  Задает исходный каталог, содержимое которого отображается в диалоговом окне для выбора файлов.

- `void setSelectedFile(File file)`
- `void setSelectedFiles(File[] file)`

  Задает файл, выбранный в окне по умолчанию.

- `void setMultiSelectionEnabled(boolean b)`

  Устанавливает или отменяет режим выбора нескольких файлов.

- `void setFileSelectionMode(int mode)`

  Позволяет пользователю выбирать только файлы (по умолчанию), только каталоги или и файлы, и каталоги одновременно. Параметр `mode` может принимать

значения JFileChooser.FILES_ONLY, JFileChooser.DIRECTORIES_ONLY и JFileChooser.FILES_AND_DIRECTORIES

- int showOpenDialog(Component parent)
- int showSaveDialog(Component parent)
- int showDialog(Component parent, String approveButtonText)

Выводит на экран диалоговое окно, в котором кнопка выбора имеет надпись Open, Save или произвольное имя, заданное строкой approveButtonText. Возвращает значения APPROVE_OPTION, CANCEL_OPTION (пользователь щелкнул на кнопке Cancel или закрыл диалоговое окно) или ERROR_OPTION (в случае возникновения ошибки).

- File getSelectedFile()
- File[] getSelectedFiles()

Возвращает файл или файлы, выбранные пользователем (или возвращает значение null, если пользователь ничего не выбрал).

- void setFileFilter(FileFilter filter)

Устанавливает маску файлов для диалогового окна. В окне отображаются только те файлы, для которых метод filter.accept() возвращает значение true.

- void addChoosableFileFilter(FileFilter filter)

Добавляет фильтр к списку.

- void setAcceptAllFileFilterUsed(boolean b)

Включает фильтр All files в раскрывающийся список или удаляет соответствующий пункт из списка.

- void resetChoosableFileFilters()

Очищает список фильтров. Если фильтр All files специально не отменен, он остается в списке.

- void setFileView(FileView view)

Устанавливает средства просмотра, предоставляющие информацию о файлах, отображаемых в окне.

- void setAccessory(JComponent component)

Задает компонент доступа.

API **javax.swing.filechooser.FileFilter 1.2**

- boolean accept(File f)

Возвращает значение true, если указанный файл должен отображаться а диалоговом окне.

- String getDescription()

Возвращает описание указанного фильтра, например Image files (*.gif, *.jpeg).

API javax.swing.fileChooser.FileView 1.2

- `String getName(File f)`

  Возвращает имя файла `f` или значение `null`. Обычно этот метод возвращает значение `f.getName()`.

- `String getDescription(File f)`

  Возвращает описание файла `f`, ориентированное на пользователя, или значение `null`. Например, если файл `f` представляет собой документ в формате HTML, этот метод может вернуть его заголовок.

- `String getTypeDescription(File f)`

  Возвращает описание типа файла `f`, ориентированное на пользователя, или значение `null`. Например, если файл `f` представляет собой документ в формате HTML, этот метод может вернуть строку `Hypertext document`.

- `Icon getIcon(File f)`

  Возвращает пиктограмму, назначенную для файла `f`, или значение `null`.

- `Boolean isTraversable(File f)`

  Если пользователь может открыть указанный каталог, возвращается значение `Boolean.TRUE`. Если каталог представляет собой составной документ, может возвращать значение `false`. Подобно методам `FileView`, этот метод может возвращать значение `null`, отмечая тот факт, что диалоговое окно должно использовать обычный стиль отображения файлов.

Диалоговое окно для выбора цвета

Очевидно, что диалоговое окно для выбора файлов — довольно сложный элемент пользовательского интерфейса, который вряд ли стоит реализовывать самостоятельно. Многие инструментальные средства для создания пользовательского интерфейса дают возможность формировать диалоговые окна для выбора даты или времени, валюты, шрифтов, цвета и т.п. Это дает двойную выгоду. В результате программисты могут использовать готовые высококачественные компоненты, а пользователи получают удобный интерфейс.

На данный момент в пакете Swing, кроме окна для выбора файлов, предусмотрено лишь одно подобное диалоговое окно, позволяющее выбирать цвет. Это окно реализовано с помощью класса `JColorChooser`. Внешний вид различных вкладок окна для выбора цвета показан на рис. 9.52–9.54. Как и класс `JFileChooser`, класс `JColorChooser` является компонентом, однако он содержит удобные методы, позволяющие создавать диалоговые окна.

Модальное диалоговое окно для выбора цвета создается с помощью следующего выражения:

```
Color selectedColor = JColorChooser.showDialog(parent,
   title, initialColor);
```

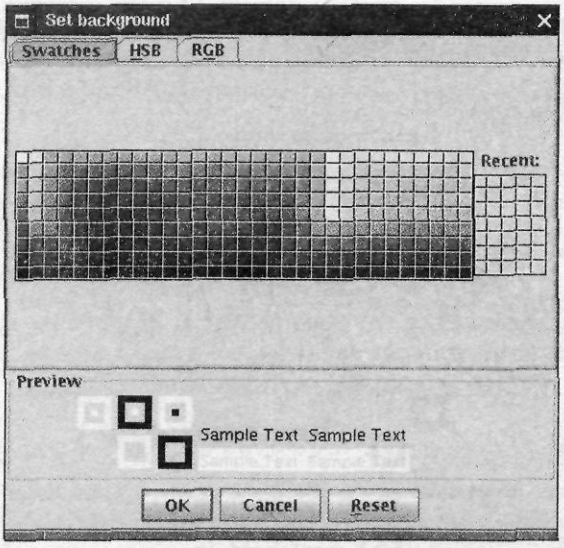


Рис. 9.52. Панель Swatches диалогового окна для выбора цвета

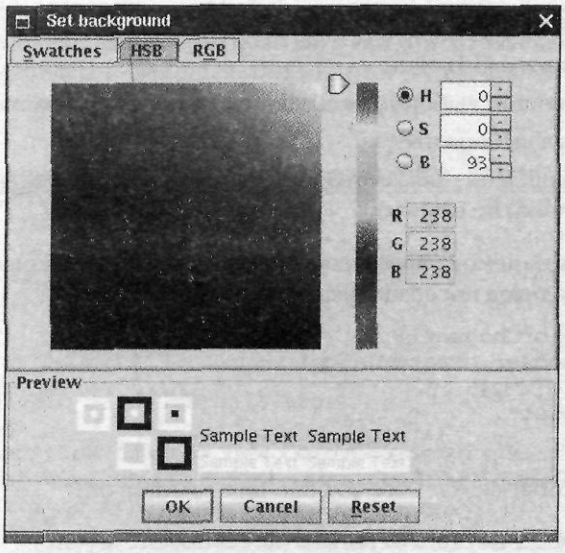


Рис. 9.53. Панель HSB диалогового окна для выбора цвета

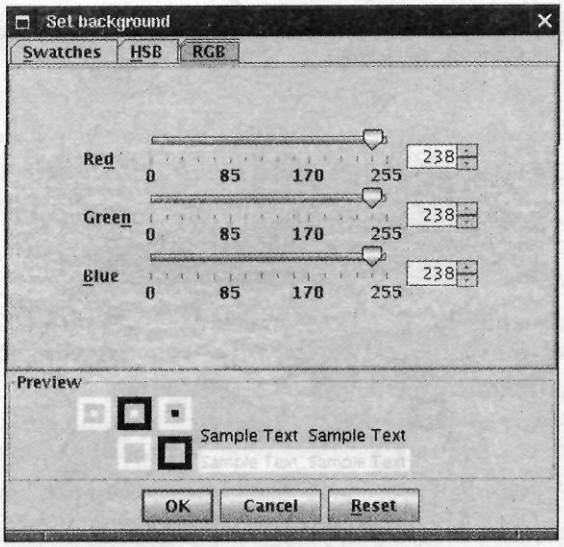


Рис. 9.54. Панель RGB диалогового окна для выбора цвета

Выбор цвета можно также обеспечить с помощью немодального диалогового окна. Для этого нужно указать следующие элементы.

- Родительский компонент.
- Заголовок диалогового окна.
- Флаг, управляющий выбором модального или немодального диалогового окна.
- Компонент для выбора цвета.
- Обработчик событий, связанных с кнопами OK и Cancel (или параметр `null`, если обработчики не нужны).

Попробуем создать немодальное диалоговое окно для установки цвета фона. Установка цвета производится после щелчка на кнопке OK.

```
chooser = new JColorChooser();
dialog = JColorChooser.createDialog(
   parent,
   "Background Color",
   false                  /* немодальное окно *.
   chooser,
   new ActionListener() // Обработчик для кнопки OK.
      {
         public void actionPerformed(ActionEvent event)
         {
            setBackground(chooser.getColor());
         }
      },
   null     /* Обработчик для кнопки Cancel не задается */);
```

Еще лучше предоставить пользователю возможность сразу увидеть эффект выбора. Для отслеживания нужна модель выбора и обработчик изменений.

```
chooser.getSelectionMode().addChangeListener (new
   ChangeListener()
   {
      public void stateChanged(ChangeEvent event)
      {
          // Действия с методом chooser.getColor();
      }
   });
```

В данном случае кнопки OK и Cancel становятся лишними. Можно просто добавить компонент для выбора цвета в немодальное диалоговое оено.

```
dialog = new JDialog(parent, false /* немодальное */);
dialog.add(chooser);
dialog.pack();
```

Программа, приведенная в листинге 9.22, показывает три типа диалога. Щелкнув на кнопке Modal, пользователь должен будет выбрать цвет и лишь затем сможет продолжить работу. Щелкнув на кнопке Modeless, пользователь создаст немодальное окно, но изменение цвета произойдет только после того, как он щелкнет на кнопке OK. И наконец, щелкнув на кнопке Immediate, пользователь создаст немодальное диалоговое окно без кнопок; как только он выберет новый цвет, цвет фона панели немедленно изменится.

На этом мы завершаем обсуждение компонентов пользовательского интерфейса. Материал, изложенный в главах 7–9, позволит вам самостоятельно создавать простые графические интерфейсы с помощью библиотеки Swing. Более подробно соответствующие вопросы обсуждаются во втором томе.

Листинг 9.22. Содержимое файла `ColorChooserText.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class ColorChooserTest
{
   public static void main(String[] args)
   {
      ColorChooserFrame frame = new ColorChooserFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с панелью для выбора цвета.
*/
class ColorChooserFrame extends JFrame
{
   public ColorChooserFrame()
```

```
   {
      setTitle("ColorChooserTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление панели для выбора цвета к фрейму.

      ColorChooserPanel panel = new ColorChooserPanel();
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}

/**
   Панель с кнопками для выбора трех типов компонентов.
*/
class ColorChooserPanel extends JPanel
{
   public ColorChooserPanel()
   {
      JButton modalButton = new JButton("Modal");
      modalButton.addActionListener(new ModalListener());
      add(modalButton);

      JButton modelessButton = new JButton("Modeless");
      modelessButton.addActionListener(new ModelessListener());
      add(modelessButton);

      JButton immediateButton = new JButton("Immediate");
      immediateButton.addActionListener(new ImmediateListener());
      add(immediateButton);
   }

   /**
      Данный обработчик отображает модальное окно
      для выбора цвета.
   */
   private class ModalListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         Color defaultColor = getBackground();
         Color selected = JColorChooser.showDialog(
            ColorChooserPanel.this,
            "Set background",
            defaultColor);
         if (selected != null) setBackground(selected);
      }
   }

   /**
      Данный обработчик отображает немодальное окно
      для выбора цвета. Цвет панели изменяется после щелчка
      на кнопке OK.
   */
```

```
private class ModelessListener implements ActionListener
{
   public ModelessListener()
   {
      chooser = new JColorChooser();
      dialog = JColorChooser.createDialog(
         ColorChooserPanel.this,
         "Background Color",
         false /* немодальное */,
         chooser,
         new ActionListener() // Обработчик для кнопки OK.
            {
               public void actionPerformed(ActionEvent event)
               {
                  setBackground(chooser.getColor());
               }
            },
         null /* Обработчик для кнопки Cancel не задан */);
   }

   public void actionPerformed(ActionEvent event)
   {
      chooser.setColor(getBackground());
      dialog.setVisible(true);
   }

   private JDialog dialog;
   private JColorChooser chooser;
}

/**
   Данный обработчик отображает немодальное окно
   для выбора цвета. Цвет панели изменяется сразу
   после выбора.
*/
private class ImmediateListener implements ActionListener
{
   public ImmediateListener()
   {
      chooser = new JColorChooser();
      chooser.getSelectionModel().addChangeListener(new
         ChangeListener()
         {
            public void stateChanged(ChangeEvent event)
            {
               setBackground(chooser.getColor());
            }
         });

      dialog = new JDialog(
         (Frame) null,
         false /* немодальное */);
      dialog.add(chooser);
      dialog.pack();
   }
```

```
    public void actionPerformed(ActionEvent event)
    {
       chooser.setColor(getBackground());
       dialog.setVisible(true);
    }

    private JDialog dialog;
    private JColorChooser chooser;
  }
}
```

API **java.swing.JColorChooser 1.2**

- `JColorChooser()`

  Создает компонент для выбора цвета, у которого исходным цветом считается *белый*.

- `Color getColor()`
- `void setColor(Color c)`

  Возвращает и устанавливает текущий цвет для компонента.

- `static Color showDialog(Component parent, String title, Color initialColor)`

  Отображает на экране модальное диалоговое окно, содержащее компонент для выбора цвета.

Параметры		
	`parent`	Компонент, поверх которого отображается диалоговое окно
	`title`	Заголовок фрейма
	`initialColor`	Исходный цвет в компоненте

- `static JDialog createDialog(Component parent, String title, boolean modal, JColorChooser chooser, ActionListener okListener, ActionListener cancelListener)`

  Создает диалоговое окно, содержащее компонент для выбора цвета.

Параметры		
	`parent`	Компонент, поверх которого отображается диалоговое окно
	`title`	Заголовок фрейма
	`modal`	Принимает значение `true`, если выполнение программы должно блокироваться, пока диалоговое окно не будет закрыто
	`chooser`	Компонент для выбора цвета, который следует добавить в диалоговое окно
	`OkListener`, `cancelListener`	Обработчики для кнопок OK и Cancel

Глава 10

Аплеты

В этой главе...

- Основные понятия
- HTML-дескрипторы и атрибуты для работы с аплетами
- Мультимедиа
- Контекст аплета
- Файлы JAR
- Упаковка приложений
- Технология Java Web Start
- Хранение параметров настройки

В предыдущих главах подробно рассматривались практически все свойства языка Java, а также основные средства программирования графических приложений. Мы считаем, что Java — прекрасный (если не совершенный) универсальный объектно-ориентированный язык, и надеемся, что читатели разделяют нашу точку зрения. Невероятная популярность этого языка в первые годы его существования объясняется способностью "оживлять" Internet. Дело в том, что на языке Java можно создавать особые программы (называемые аплетами), которые загружаются из глобальной сети и выполняются браузерами. Они предназначались для того, чтобы освободить пользователей от утомительного процесса инсталлирования программного обеспечения, предоставив им доступ к программам с любого компьютера, поддерживающего язык Java, или устройства, имеющего выход во всемирную сеть.

По разным причинам аплеты так и не оправдали ожидания. Недавно компания Sun разработала альтернативный подход к доставке приложений, основанных на использовании сети. Эта технология называется *Java Web Start*. В ней исправлены некоторые из недостатков, характерные для аплетов.

В этой главе показано, как создавать и доставлять аплеты, как запаковывать аплеты и приложения для доставки пользователям. Здесь также будут обсуждаться вопросы хранения информации о конфигурации и доступа к ней.

Основные понятия

Для описания Web-страниц используется язык HTML (HyperText Markup Language — язык гипертекстов разметки). Он представляет собой набор конструкций, позволяющих описывать элементы, расположенные на странице, и формировать гипертекстовые ссылки. Например, дескриптор `<title>` обозначает заголовок страницы, которым является любой текст, помещенный после этого дескриптора. Конец заголовка обозначается дескриптором `</title>`. (Это одно из основных правил, касающихся дескрипторов: косая черта перед именем элемента означает конец соответствующего элемента.)

Основная идея, положенная в основу использования аплетов на Web-страницах, довольно проста: страница, описанная на языке HTML, должна сообщать браузеру, какой аплет надо загрузить и где его разместить. Нетрудно догадаться, что дескриптор, применяемый для работы с аплетами, должен сообщать браузеру следующую информацию.

- Откуда получить файлы классов.
- Как расположить аплет на странице (т.е. указать его размеры, координаты и т.д.).

Браузер загружает файлы с классами по сети (с локального диска) и автоматически запускает аплет на выполнение, используя встроенную виртуальную машину Java (Java Virtual Machine).

Помимо аплетов, Web-страница может содержать все остальные элементы языка HTML: маркированные или нумерованные списки, графические изображения и т.д. *Аплеты — это всего лишь часть гипертекстовой страницы. Всегда следует иметь в виду, что язык Java не является средством для разработки гипертекстовых страниц.* Это инструмент, позволяющий их "оживлять". Отсюда вовсе не следует, что элементы графического пользовательского интерфейса в аплете, написанном на языке Java, не важны. Однако эти элементы должны быть согласованы с остальными элементами Web-страницы, *созданными с помощью языка HTML*.

Мы не будем обсуждать HTML-дескрипторы, предполагая, что вы знаете основы этого языка. Для создания аплетов на языке Java нужны лишь несколько специальных дескрипторов языка HTML. Их мы, конечно, рассмотрим.

Когда появились аплеты, программисты должны были использовать для просмотра Web-страниц браузер HotJava, специально созданный компанией Sun для поддержки аплетов. Популярность аплетов существенно возросла, когда компания Netscape включила виртуальную машину языка Java в свой браузер. Вскоре появился браузер Internet Explorer. К сожалению, возникла проблема. Компания Netscape не стала поддерживать более современные версии языка Java, а компания Microsoft склонялась к решению вовсе отказаться от языка Java.

Для того чтобы решить эту проблему, компания Sun разработала и распространила инструмент под названием Java Plug-In. Используя различные механизмы расширения возможностей Internet Explorer и Netscape, этот инструмент можно совершенно свободно встроить в оба браузера. Это позволяет использовать их для запуска аплетов с помощью среды поддержки Java (Java Runtime Environment), разработанной компанией Sun. Предполагалось, что Java Plug-In позволит воспользоваться преимуществами последних версий виртуальных машин.

Чтобы запустить аплеты, код которых приведен в данной главе, с помощью своего браузера, необходимо инсталлировать текущую версию программы Java Plug-In и убедиться в том, что браузер связан с ней. Java Plug-In можно скопировать, обратившись по адресу `http://java.sun.com/getjava/`. Там же вы найдете инструкции по инсталляции данного продукта.

Считается, что, разрабатывая Web-страницы для широкой аудитории, не имеет смысла рекомендовать пользователю инсталлировать программу Java Plug-In, которая, кстати, довольно долго загружается. Более того, принимая решение о создании аплетов, надо сначала подумать, нельзя ли реализовать те же возможности средствами HTML, JavaScript либо с помощью анимационных GIF-файлов. Сложные же вычисления желательно выполнять на сервере. Для этой цели стоит использовать Java-сервлеты и JavaServer Pages.

С другой стороны, если вы разработали очень сложную программу на языке Java, задайте себе вопрос, стоит ли применять браузер в качестве средства ее доставки. Может быть, лучше создать обычное приложение, которое пользователь будет самостоятельно запускать на своей машине. Вы по-прежнему сможете использовать такие преимущества языка Java, как независимость от конкретной платформы, механизм защиты, средства доступа к базам данных и поддержку работы в сети.

Конечно, распространение программ средствами Web имеет несомненные преимущества. Для пользователя зачастую проще разместить приложение в сети, чем в локальной файловой системе. (Это особенно касается приложений, которые используются не часто.) С точки зрения администратора, поддерживать и обновлять приложение, хранящееся на сервере, легче, чем программу на локальном диске.

Таким образом, на языке Java лучше всего писать приложения, работающие в сетях intranet. Например, многие компании размещают в корпоративных сетях программы для финансовых расчетов, обработки заказов и т.д. Эти программы относительно невелики, должны взаимодействовать с базами данных, удовлетворять специфические потребности конкретной компании и обладать гибкостью, которую не могут предоставить обычные Web-формы. Аплеты и механизм Java Web Start превосходно подходят для доставки таких программ. Поскольку круг пользователей корпоративной сети ограничен, проблемы с инсталлированием среды поддержки выполнения программ на языке Java обычно не возникает.

Итак, учитывая роль, которую играют аплеты в технологии Java Web Start, начнем именно с них.

Простой аплет

По традиции начнем с простейшего аплета. Перепишем в виде аплета программу `NotHelloWord`. Прежде чем сделать это, нужно пояснить, что, с точки зрения программиста, аплет не представляет собой ничего удивительного. Это лишь подкласс класса `java.applet.Applet`. Отметим, что, хотя пакет `java.applet` не входит в пакет `AWT`, аплет представляет собой AWT-компонент. Соответствующий фрагмент иерархии наследования классов показан на рис. 10.1. В рамках этой книги для реализации аплетов мы будем пользоваться библиотекой Swing. Все наши аплеты будут расширять класс `JApplet`, являющийся суперклассом всех аплетов, создаваемых с помощью библиотеки Swing. Как показано на рис. 10.1, класс `JApplet` является подклассом обычного класса `Applet`.

Если аплет содержит компоненты библиотеки Swing, следует расширять класс `JApplet`. Компоненты Swing внутри объекта обычного класса `Applet` будут отображаться неправильно.

В листинге 10.1 показан код аплета, выполняющего те же действия, что и программа Not Hello World.

Обратите внимание на сходство этого аплета с программой, приведенной в главе 7. Однако, поскольку аплет существует внутри Web-страницы, здесь нет необходимости указывать метод `main()`.

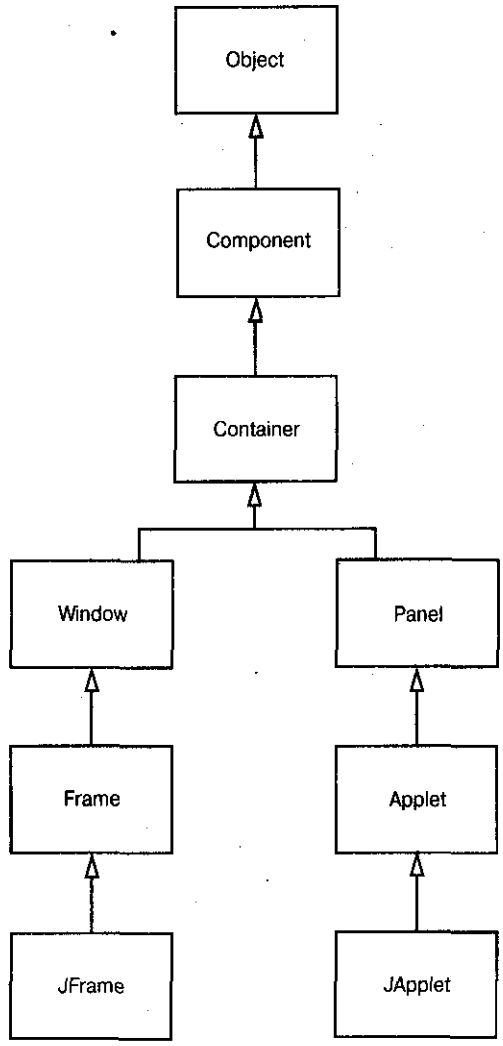


Рис. 10.1. Иерархия наследования для аплетов

Листинг 10.1. Содержимое файла NotHelloWorldApplet.java

```
/*
Для выполнения аплета в среде браузера нужен
следующий HTML-элемент
  <applet code="NotHelloWorldApplet.class"
    width="300" height="100">
  </applet>
*/

import javax.swing.*;

public class NotHelloWorldApplet extends JApplet
{
   public void init()
   {
      JLabel label = new JLabel("Not a Hello, World applet",
         SwingConstants.CENTER);
      add(label);
   }
}
```

Выполнение аплета

Для того чтобы выполнить аплет, необходимо сделать следующее.

1. Скомпилировать исходные файлы в файлы классов.
2. Создать HTML-файл, сообщающий браузеру, какой файл класса следует загрузить первым и какие размеры имеет аплет.

Файл в формате HTML принято (но не обязательно) называть так же, как и класс аплета, содержащийся внутри него. Следуя этой традиции, дадим нашему файлу имя `NotHelloWorldApplet.html`. Ниже приводится его содержимое.

```
<applet code="NotHelloWorldApplet.class" width=300 height=300>
</applet>
```

Перед тем как просмотреть аплет в браузере, желательно проверить его с помощью программы `appletviewer`, предназначенной для просмотра аплетов. Эта программа входит в пакет JDK. Чтобы запустить аплет с помощью `appletviewer`, нужно выполнить команду

```
appletviewer NotHelloWorldApplet.html
```

В командной строке указывается имя HTML-файла, а не файла класса. На рис. 10.2 показано окно программы `appletviewer`, в котором отображается аплет `NotHelloWorldApplet`.

Аплеты можно также запускать на выполнение, находясь в текстовом редакторе или в интегрированной среде разработки. В редакторе Emacs выберите пункт меню JDE⇨Run Applet. В Eclipse выберите Run⇨Run as⇨Java Applet.

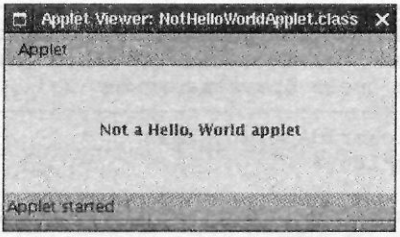


Рис. 10.2. Просмотр аплета с помощью программы appletviewer

Избежать создания дополнительного файла в формате HTML можно с помощью следующего приема. Добавьте дескриптор аплета в исходный файл *в виде комментариев*.

```
/*
  <applet code="MyApplet.class" width=300 height=300>
  </applet>
*/
public class MyApplet extends JApplet
...
```

Затем запустите программу `appletviewer`, задав в командной строке в качестве параметра *исходный файл*.

```
appletviewer NotHeloWorldApplet.java
```

Этот прием нельзя рекомендовать для постоянного использования, но он может оказаться удобным, если вы хотите минимизировать количество файлов, за которыми нужно следить во время отладки.

Программа `appletviewer` удобна на первых этапах тестирования, но в какой-то момент вам понадобится запустить аплет в браузере, чтобы почувствовать себя на месте пользователя. В частности, программа `appletviewer` показывает на экране только аплет без окружающего текста. Если файл в формате HTML содержит много аплетов, то программа `appletviewer` откроет на экране несколько отдельных окон.

Для просмотра аплета необходим браузер, поддерживающий язык Java 2. После инсталляции и конфигурации программы Java Plug-In достаточно загрузить HTML-файл в браузер (рис. 10.3). Если аплет не отображается, возможно, браузер использует встроенную виртуальную машину. Чтобы решить эту проблему, вам придется настроить его на взаимодействие с программой Java Plug-In.

Если вы изменили аплет и скомпилировали его снова, вам нужно перезапустить браузер, чтобы он смог загрузить новые файлы классов. Простое обновление HTML-страницы не приводит к загрузке нового кода. Утомительной процедуры повторного запуска браузера можно избежать, открыв консоль Java и введя команду *x*, которая очищает кэш-память загрузчика классов. После этого вы можете повторно открывать HTML-файл, и на этот раз будет воспроизведен новый код аплета. В браузерах Netscape и Mozilla консоль Java запускается из меню. Если вы работаете на компьютере под управлением операционной системы Windows, щелкните на пиктограмме Java Plug-In, расположенной на панели управления, и введите команду вывода на экран консоли Java.

Если вы работаете с более старыми браузерами (например, Netscape 4), замените дескрипторы `<applet>` специальными дескрипторами `<object>` или `<embed>`. Это заставит браузер загрузить Java Plug-In. Данный процесс описан на Web-странице `http://java.sun.com/j2se/5.0/docs/guide/plugin/developer_guide/html_converter.html`. Кроме того, с нее можно загрузить инструментальные средства, автоматизирующие преобразование Web-страниц. При работе с более современными браузерами преобразование Web-страниц не требуется.

Рис. 10.3. Просмотр аплета с помощью браузера

Преобразование приложения в аплет

Приложение с графическим интерфейсом, использующее библиотеку AWT (и допускающее запуск из командной строки) легко преобразовать в аплет и встроить в Web-страницу. Особенно важно, что при этом весь код пользовательского интерфейса можно оставить без изменений.

Для того чтобы выполнить данную задачу, нужно сделать следующее.

1. Создать HTML-страницу с соответствующим дескриптором для загрузки кода аплета.
2. Определить подкласс класса `JApplet`. Этот класс должен быть общедоступным (`public`), иначе аплет будет невозможно загрузить.
3. Удалить из приложения метод `main()`. Окно фрейма для приложения создавать не следует. Приложение будет отображаться в браузере.
4. Перенести все операторы инициализации из конструктора окна фрейма в метод аплета `init()`. Создавать объект аплета не обязательно — браузер сам создаст его и вызовет метод `init()`.
5. Удалить вызов метода `setSize()`. В аплетах размеры задаются в HTML-файле с помощью параметров `width` и `height`.
6. Удалить вызов метода `setDefaultCloseOperation()`. Аплет нельзя закрыть; он прекращает работу вместе с браузером.

7. Если в приложении есть вызов метода `setTitle()`, следует его удалить. Аплеты не имеют заголовков. (Разумеется, можно задать заголовок самой Web-страницы, используя дескриптор `<title>` языка HTML.)
8. *Не вызывать метод* `setVisible(true)`. *Аплет отображается на экране автоматически.*

Далее в этой главе вы узнаете, как можно создать программу, являющуюся одновременно и аплетом, и приложением.

В качестве примера рассмотрим превращение приложения `Calculator` (см. главу 7). На рис. 10.4 показано, как выглядит этот аплет на Web-странице.

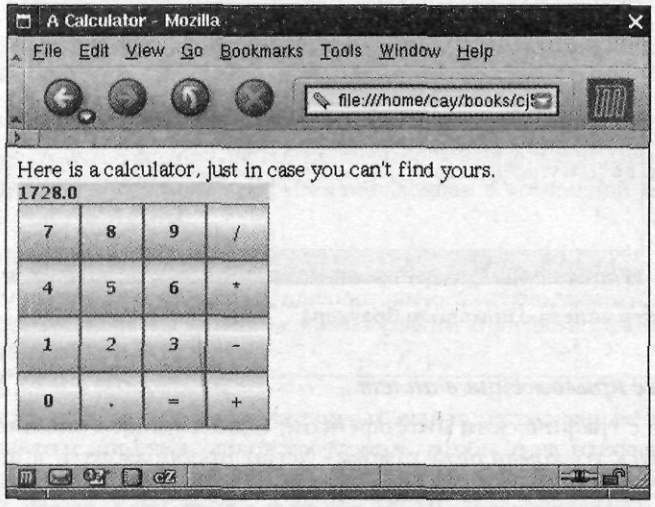


Рис. 10.4. Аплет, реализующий калькулятор

В листинге 10.2 показана HTML-страница, содержащая аплет. Обратите внимание на то, что кроме дескриптора, ссылающегося на аплет, в ней содержится текстовая информация.

Листинг 10.2. Содержимое файла `Calculator.html`

```
<html>
   <head><title>A Calculator</title></head>
   <body>
      <p>Этот калькулятор пригодится, если вашего
      нет под рукой.</p>
      <applet code="CalculatorApplet.class"
         width="180" height="180">
      </applet>
   </body>
</html>
```

Код аплета приведен в листинге 10.3. Мы определяем подкласс класса `JApplet`, а в метод `init` добавляем код инициализации. Кроме того, из программы удалены вызовы методов `setTitle()`, `setSize()`, `setDefaultCloseOperation()` и `setVisible()`. Класс `CalculatorPanel` оставлен без изменения, а его код пропущен. Код класса см. в главе 9.

Листинг 10.3. Содержимое файла `CalculatorApplet.java`

```
import java.awt.*;
import javax.swing.*;

public class CalculatorApplet extends JApplet
{
   public void init()
   {
      CalculatorPanel panel = new CalculatorPanel();
      add(panel);
   }
}
```

`java.applet.Applet` 1.0

- `void init()`

  Вызывается при первой загрузке аплета. Переопределите этот метод и разместите в нем весь код, выполняющий инициализацию.

- `void resize(int width, int height)`

  Задает новые размеры аплета. Если бы этот метод влиял на аплеты, отображаемые на Web-страницах, он был бы очень удобен. К сожалению, с современными браузерами этот метод не работает, поскольку конфликтует с их механизмами компоновки.

Жизненный цикл аплета

Основу для создания любого аплета составляют четыре метода из класса `Applet`: `init()`, `start()`, `stop()` и `destroy()`. Ниже приводится краткое описание, в котором указано, когда их следует вызывать и какой код они должны содержать.

- `init()`

  Этот метод используется для инициализации аплета. Он очень похож на конструктор, поскольку автоматически вызывается при первом запуске аплета. Обычно все действия этого метода внутри аплета сводятся к обработке значений `<param>` (обработка параметров будет рассмотрена далее в этой главе) и добавлению компонентов пользовательского интерфейса. Несмотря на то что аплеты могут иметь конструктор, инициализацию следует выполнять с помощью метода `init()`.

- `start()`

  Этот метод автоматически вызывается *после* метода `init()`. Он также вызывается, если пользователь вернулся на страницу, содержащую аплет, после просмотра других страниц. Это значит, что, в отличие от метода `init()`, `start()`

можно вызывать повторно. По этой причине код, который должен выполняться только один раз, следует помещать в метод `init()`, а не в `start()`. Этот метод следует вызывать при повторном запуске потока для аплета, например, для возобновления анимации. Если аплет не выполняет никаких действий, которые нужно приостановить при переходе пользователя на другую страницу и возобновить при возврате, метод `start()` переопределять не нужно (как и метод `stop()`).

- `stop()`

  Этот метод вызывается автоматически, когда пользователь покидает страницу, на которой расположен аплет. Таким образом, в одном и том же аплете он может выполняться несколько раз. Метод `stop()` предназначен для остановки процессов, на выполнение которых затрачивается слишком много времени. Это предотвращает замедление работы системы в те моменты, когда пользователь не работает с аплетом. Этот метод не следует вызывать явно. Если аплет не воспроизводит анимационные последовательности или аудиофайлы, метод `stop()` можно не переопределять.

- `destroy()`

  Этот метод вызывается только при нормальном завершении работы браузера. Поскольку аплеты предназначены для использования в составе HTML-страниц, заботиться об освобождении ресурсов после закрытия страницы, содержащей аплет, совершенно не обязательно. Если все же вы решили сделать это самостоятельно, то вам придется переопределить метод `destroy()`.

API **`java.applet.Applet 1.0`**

- `void start()`

  В этом методе размещается код, который подлежит выполнению *каждый раз*, когда пользователь посещает страницу браузера, содержащую данный аплет. Обычно этот метод повторно активизирует поток.

- `void stop()`

  В этом методе размещается код, который подлежит выполнению *каждый раз*, когда пользователь покидает страницу браузера, содержащую данный аплет. Обычно этот метод приостанавливает поток.

- `void destroy()`

  Переопределите этот метод, реализовав в нем код, который подлежит выполнению при выходе из браузера.

Обеспечение безопасности при работе аплета

Поскольку аплеты предназначены для загрузки с удаленных Web-страниц и последующего выполнения на локальном компьютере, вопросы безопасности крайне важны. Браузер, поддерживающий язык Java, может загрузить код аплета и немедленно его выполнить. Пользователь не имеет никакой возможности вмешаться в данный процесс. По этой причине аплеты, в отличие от приложений, ограничены в своих возможностях. Как только аплет пытается нарушить правила доступа, *диспетчер защиты* генерирует исключение `SecurityException`. (Более подробно диспетчеры защиты рассматриваются в томе 2.)

Что аплеты *могут* делать на всех платформах? Аплет может выводить на экран графические изображения, воспроизводить звуковые данные, воспринимать нажатия клавиш и щелчки мышью, а также взаимодействовать с компьютером, с которого он был загружен. Этого вполне достаточно для того, чтобы передавать информацию пользователю и получать от него данные. Среда, ограничивающая выполнение аплетов, часто называется "песочницей" (sandbox). Аплеты, "играющие в песочнице", не могут изменять систему пользователя или шпионить за ней.

В частности, при выполнении аплета действуют следующие правила.

- Аплеты *ни при каких обстоятельствах* не могут запускать исполняемые файлы на локальном компьютере.
- Аплет не может взаимодействовать ни с одним компьютером, кроме сервера, с которого он был загружен. Это правило часто формулируют так: "Аплет может звонить только домой". Это защищает пользователя от аплетов, пытающихся передавать вовне информацию из корпоративной сети.
- Аплет не может считывать данные из локальной файловой системы или записывать их туда.
- Аплеты не способны получать информацию о локальном компьютере, за исключением номера версии языка Java, имени и версии операционной системы, а также символов, используемых для разделения имен каталогов в пути файлов (например, `\`, `/`, `:` или `;`) и символов завершения строк (например, `\n` или `\r\n`). В частности, аплеты не могут определять имя пользователя, адрес его электронной почты и получать другую подобную информацию.
- Все окна, открываемые аплетом, сопровождаются предупреждением.

Выполнение этих правил возможно лишь потому, что аплеты *интерпретируются* виртуальной машиной, а не выполняются центральным процессором пользовательского компьютера. Поскольку интерпретатор проверяет все команды и области программы, аплет, созданный злоумышленником (или просто небрежно написанный), никогда не сможет разрушить операционную систему, модифицировать содержимое системной памяти или изменить привилегии, предоставленные операционной системой.

В некоторых ситуациях эти ограничения оказываются слишком строгими. Например, в корпоративной сети легко представить себе аплет, которому нужен доступ к локальным файлам. Для того чтобы в разных случаях применять разные ограничения, можно использовать *аплеты с электронной подписью*. Такие аплеты сопровождаются сертификатом, подтверждающим личность отправителя. Криптографические методы гарантируют, что этот сертификат не может быть подделан. Если вы доверяете человеку, подписавшему аплет, то можете предоставить его аплету дополнительные права. (Электронные подписи рассматриваются в томе 2.)

Если вы уверены в аплете, браузер может предоставить ему дополнительные возможности. Например, аплетам, подписанным сотрудниками вашей корпорации, вы можете доверять больше, чем аплетам, полученным с Web-узла `www.hacker.com`. Настраиваемая система безопасности языка Java предоставляет различные уровней доступа. Аплетам, которым вы доверяете полностью, можно предоставить права локальных приложений. Программы от сомнительных отправителей могут получить ограниченные привилегии. Действия же совершенно незнакомых аплетов надо максимально ограничить.

Приложения, созданные по технологии Java Web Start (она будет рассмотрена далее), имеют более гибкий механизм обеспечения безопасности. Они получают доступ к некоторым системным ресурсам с согласия пользователя.

Итак, язык Java имеет три отдельных механизма для обеспечения безопасности.

1. Код программы интерпретируется виртуальной машиной, а не выполняется непосредственно.
2. Диспетчер защиты отслеживает все потенциально опасные операции.
3. Аплеты могут сопровождаться электронной подписью, подтверждающей их происхождение.

В противоположность аплетам, модель безопасности в технологии ActiveX, разработанной Microsoft, опирается исключительно на электронную подпись. Для того чтобы запустить компонент ActiveX, вы должны полностью доверять ему. Если у вас несколько надежных поставщиков, эта модель работает прекрасно, но сферу ее применения невозможно расширить на всю Web. Используя браузер Internet Explorer, вы можете увидеть технологию ActiveX в действии. Для того чтобы инсталлировать программу Java Plug-In для работы с браузером Internet Explorer, нужно получить сертификат компании Sun, который сообщает, что данный код создан именно в компании Sun. В нем ничего не говорится о качестве кода. После начала инсталляции пользователь никак не может повлиять на уровень безопасности этой программы.

Создание окон с помощью аплетов

Аплет встраивается в Web-страницу, находясь внутри фрейма. Размеры этого фрейма задаются параметрами `width` и `height`, указанными в дескрипторах аплета в HTML-файле. Часто это бывает довольно неудобно. Многие программисты интересуются: можно ли создавать окна по мере необходимости, чтобы сэкономить пространство? Да, можно. В данном разделе мы рассмотрим пример простого аплета с единственной кнопкой Calculator. После щелчка на этой кнопке калькулятор отображается на экране в отдельном окне.

Чтобы создать окно, надо использовать класс `JFrame`, но при этом не вызывайте метод `setDefaultCloseOperation()`.

```
frame = new CalculatorFrame();
frame.setTitle("Calculator");
frame.setSize(200, 200);
```

После щелчка на кнопке Calculator на экране отображается диалоговое окно. После повторного щелчка калькулятор исчезает.

```
JButton calcButton = new JButton("Calculator");
calcButton.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent evt)
      {
         frame.setVisible(!frame.isVisible());
      }
   });
```

Существует одна особенность, о которой вам следует знать, прежде чем поместить аплет на свою Web-страницу. Для того чтобы увидеть, как будет выглядеть калькулятор на компьютере пользователя, используйте для просмотра соответствующей Web-страницы браузер, а не программу `appletviewer`. Вы увидите, что калькулятор будет окружен рамкой, содержащей предупреждающее сообщение (рис. 10.5).

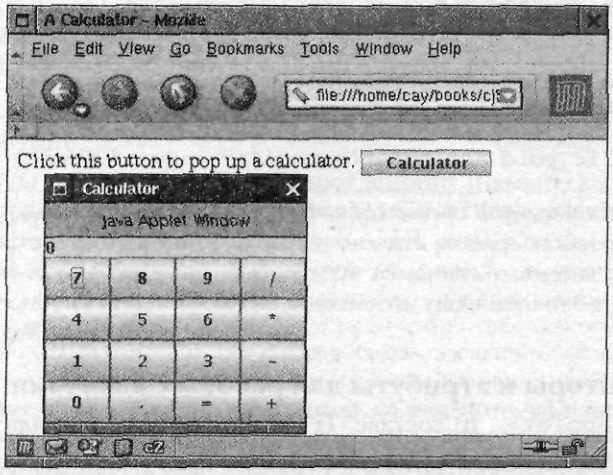


Рис. 10.5. Окно калькулятора при просмотре Web-страницы с помощью браузера

В ранних версиях браузеров это сообщение выглядело зловеще: `Untrusted Java Applet Window`. В каждой последней версии JDK оно немного смягчалось `Unauthenticated Java Applet Window`, `Warning: Java Applet Window`. Теперь это сообщение выглядит так: `Java Applet Window`.

Подобное сообщение характерно для всех Web-браузеров. Браузер принимает меры к тому, чтобы пользователь не спутал окно аплета с окном локального приложения и не ввел в нем конфиденциальные данные, например имя пользователя и пароль.

Для того чтобы предотвратить незаконный сбор информации окна, открываемые аплетом сопровождаются предупреждением. При желании вы можете настроить Java Plug-in так, чтобы для аплетов с электронной подписью предупреждение не выводилось.

В листинге 10.4 приведен код класса `PopupCalculatorApplet`. Класс `CalculatorPanel` из главы 7 остался без изменения и здесь не приводится.

Листинг 10.4. Содержимое файла `PopupCalculatorApplet.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PopupCalculatorApplet extends JApplet
{
   public void init()
   {
      // Создание фрейма с панелью калькулятора.
```

```
      final JFrame frame = new JFrame();
      frame.setTitle("Calculator");
      frame.setSize(200, 200);
      frame.add(new CalculatorPanel());

      // Кнопка для вызова или удаления калькулятора.

      JButton calcButton = new JButton("Calculator");
      add(calcButton);

      calcButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               frame.setVisible(!frame.isVisible());
            }
         });
   }
}
```

HTML-дескрипторы и атрибуты для работы с аплетами

Дескриптор <applet> в составе HTML-документа выглядит приблизительно следующим образом:

```
<applet code>="NotHelloWorldApplet.class" width=100 height=100>
```

Как уже было сказано ранее, атрибут code задает имя класса; значение этого атрибута представляет собой имя файла, оканчивающееся расширением .class. Атрибуты width и height определяют размеры окна, в котором будет размещен аплет. Значения обоих параметров задаются в пикселях. Нужно также указать закрывающий дескриптор </applet>. Текст, размещенный между дескрипторами <applet> и </applet>, выводится на экран в том случае, если браузер не может отобразить аплет. Атрибуты code width и height необходимы. Если хотя бы один из них будет пропущен, браузер не сможет загрузить аплет.

Дескриптор <applet> включают в HTML-страницу, код которой может выглядеть так, как показано ниже.

```
<head>
      <title>NotHelloWorldApplet</title>
   </head>
   <body>
      <p>Следующая строка текста выводится под управлением
         системы Java:</p>
      <applet code="NotHelloWorldApplet.class"
       width="100" height="100">
         Если ваш браузер поддерживает Java, на этом
         месте будет окно аплета.
      </applet>
   </body>
</html>
```

В соответствии со спецификацией языка HTML дескрипторы (например, <applet>) и их атрибуты могут быть заданы как прописными, так и строчными буквами. То же относится и к имени класса. Однако в ряде случаев регистр символов может оказаться важным, например в некоторых системах он учитывается в именах JAR-файлов.

Атрибуты, задающие расположение аплетов

Ниже приводится краткое описание различных атрибутов, которые можно (и нужно) использовать в дескрипторе <applet> для того, чтобы задать местоположение аплета.

- `widtht, height`

  Эти атрибуты необходимы. Они задают ширину и высоту окна аплета в пикселях. В программе `appletviewer` эти параметры определяют первоначальный размер аплета. Программист может изменять размер любого окна, созданного программой `appletviewer`. В браузере изменить размеры аплета *невозможно*. Поэтому, включая аплет в состав документа, надо заранее продумать, сколько места понадобится аплету.

- `align`

  Этот атрибут задает вид выравнивания аплета. Есть два основных варианта. Аплет может представлять собой блок, окруженный текстом со всех сторон, или располагаться в составе строки, т.е рассматриваться как большой символ. Два значения этого атрибута (`left` и `right`) задают обтекание аплета текстом. Остальные значения вставляют аплет непосредственно в текст.

  Допустимые значения атрибута `align` показаны в табл. 10.1.

Таблица 10.1. Средства, определяющие расположение аплета

Атрибут	Описание
`left`	Помещает аплет возле левой границы. Текст размещается справа от аплета
`right`	Помещает аплет возле правой границы. Текст размещается слева от аплета
`bottom`	Выравнивает нижнюю границу аплета по нижней границе текста в текущей строке
`top`	Выравнивает верхнюю границу аплета по верхней границе текущей строки
`texttop`	Выравнивает верхнюю границу аплета по верхней границе текста в текущей строке
`middle`	Выравнивает середину аплета по базовой линии текущей строки
`absmiddle`	Выравнивает середину аплета по середине текущей строки
`baseline`	Выравнивает нижнюю границу аплета по базовой линии текущей строки
`absbottom`	Выравнивает верхнюю границу аплета по нижней границе текущей строки
`vspace, hspace`	Необязательные атрибуты, указывающие количество пикселей над и под аплетом (`vspace`), а также по бокам (`hspace`)

На рис. 10.6 показаны варианты выравнивания аплетов по отношению к окружающему его тексту. Вертикальная черта в начале каждой строки представляет изображение. Поскольку оно выше, чем текст, по нему можно судить о разнице между выравниванием по нижней и верхней границе строки и нижней и верхней границе текста.

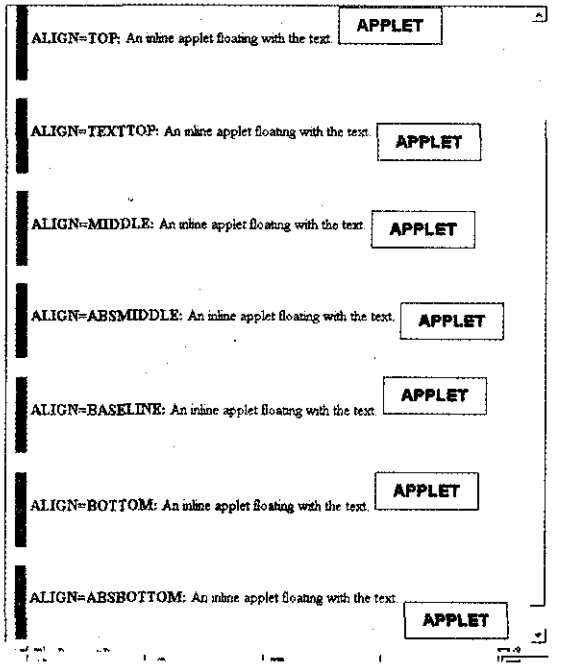


Рис. 10.6. Выравнивание аплетов

Атрибуты для указания кода аплета

Описанные ниже атрибуты сообщают браузеру, как найти код аплета.

- `code`

  Данный атрибут задает имя файла, содержащего класс аплета. Расположение файла определяется относительно пути, заданного посредством атрибута `codebase` см. ниже). Если атрибут `codebase` отсутствует, расположение файла вычисляется относительно каталога, содержащего HTML-документ. Если используется относительный путь, он должен отражать пакет, в котором находится класс аплета. Например, если класс аплета находится в пакете `com.mycompany`, то задается атрибут `code="com/mycompany/MyApplet.class"` или `code="com.mycompany.MyApplet.class"`. Атрибут `codebase` позволяет ссылаться на файл, расположенный в любой позиции файловой системы.

  Атрибут `code` задает только имя файла, содержащего класс аплета. Разумеется, аплет может содержать и другие файлы классов. Загружая класс, содержащий аплет, браузер распознает, какие файлы нужны еще, и загружает их дополнительно.

- `codebase`

  Этот необязательный атрибут информирует браузер о том, что файлы классов находятся в каталоге, указанном в качестве значения этого атрибута. По желанию вы можете указывать даже абсолютный URL каталога на другом сервере. Однако гораздо чаще задается относительный URL. Предположим, что структура каталогов выглядит так, как показано ниже.

  ```
  aDirectory/
     ├─ MyPage.html
     └─ myApplets/
           └─ CalculatorApplet.class
  ```

  В этом случае в документе `MyPage.html` может присутствовать следующий дескриптор:

  ```
  <applet code=CalculatorApplet.class" codebase="MyApplets"
          WIDTH=100 height=150>
  ```

- `archive`

  Этот необязательный атрибут перечисляет архивные файлы Java или файлы, содержащие классы и другие ресурсы для аплета (см. раздел, посвященный JAR-файлам). Такие файлы загружаются с сервера перед загрузкой аплета. Подобная технология значительно ускоряет работу, поскольку для загрузки JAR-файла, содержащего много мелких файлов, необходим лишь один HTML-запрос. В списке JAR-файлы разделяются запятыми.

  ```
  <applet code="CalculatorAPplet.class" archive=
    "CalculatorClasses.jar,corejava/CoreJavaClasses.jar"
    width=100 height=150>
  ```

- `object`

  Этот дескриптор позволяет задать имя файла, содержащего *сериализованный объект аплета*. (Объект называется *сериализованным*, если все его поля данных записаны в отдельном файле. Сериализация обсуждается в главе 12.) Перед отображением аплета объект десериализуется. Если в дескрипторе присутствует этот атрибут, метод `init()` не вызывается; выполнение аплета начинается с метода `start()`. Перед сериализацией объекта аплета необходимо вызвать его метод `stop`. Благодаря сериализации создается возможность реализовывать браузеры с сохранением состояния, в которых после повторной загрузки восстанавливается то состояние аплета, которое было в момент завершения предыдущего сеанса работы. К сожалению, эта возможность практически не используется разработчиками Web-страниц.

  В каждом дескрипторе `<applet>` должен содержаться атрибут `code` либо `object`.

  ```
  <applet object="CalculatorApplet.ser" width=100 height=150>
  ```

- `name`

  С помощью атрибута `name` аплету присваивается имя, которое затем можно использовать в сценарии. Браузеры Netscape и Internet Explorer позволяют вызывать методы аплета, размещенного на странице, средствами JavaScript. В этой

книге вопросы, связанные с языком JavaScript, не рассматриваются, поэтому мы лишь вкратце опишем, как программа на языке Java вызывается из JavaScript-сценария.

JavaScript представляет собой язык для разработки сценариев, которые можно использовать на Web-странице. Он разработан компанией Netscape и вначале назывался LiveScript. Этот язык никак не связан с языком Java. Название JavaScript было продиктовано рекламными соображениями. Подмножество этого языка под названием ECMAScript было стандартизовано и получило название AECMA-262. Однако, что совершенно неудивительно, компании Netscape и Microsoft стали поддерживать в своих *браузерах несовместимые между собой расширения этого стандарта.*

Для того чтобы получить доступ к аплету из сценария на языке JavaScript, сначала нужно дать ему имя.

```
<applet code="CalculatorApplet.class"
   width=100 height=150
   name="calc">
</applet>
```

Теперь этот объект доступен с помощью ссылки `document.applets.`*имя_аплета*.

```
var calcApplet = document.applets.calc;
```

Используя интеграцию между языками Java и JavaScript, поддерживаемую браузерами Netscape и InternetExplorer, *можно вызывать методы аплета.*

```
calcApplet.clear();
```

(Наш аплет `CalculatorApplet` не имеет метода `clear()`. Приведенный выше вызов использовался лишь для иллюстрации синтаксиса.)

Атрибут `name` приобретает большое значение, если вы хотите поместить два взаимодействующих аплета на одну и ту же страницу. Сначала нужно присвоить имя каждому аплету. Затем следует передать строку с именем методу `getApplet()` класса `AppletComplex`. *Механизм взаимодействия аплетов обсуждается далее в этой главе.*

В документе, расположенном по адресу http://www.javaworld.com/javatips/jw-javatip80.html, описывается, как можно использовать средства взаимодействия JavaScript и Java для решения классической проблемы: изменения размеров окна аплета, заданных с помощью *атрибутов width и height. Разобрав приведенный пример, можно сделать вывод, насколько* запутанным может быть JavaScript-код, предназначенный для работы на различных браузерах.

Атрибуты для браузеров, поддерживающих Java

Если Web-страница, содержащая дескриптор `<applet>`, обрабатывается браузером, *не поддерживающим Java-аплеты, то этот браузер проигнорирует неизвестные* ему дескрипторы `applet` и `param`. Весь текст, размещенный между дескрипторами `<applet>` и `</applet>`, будет выведен на экран. И наоборот, браузеры, поддерживающие язык Java, не отображают на экране текст, находящийся между `<applet>`

и </applet>. Таким образом, текст между данными дескрипторами ориентирован в основном на пользователей, работающих с очень старыми версиями браузеров.

```
<applet code="CalculatorApplet.class" width=100 height=150>
Если бы ваш браузер поддерживал язык Java, здесь вы увидели бы калькулятор.
</applet>
```

Разумеется, большинство современных браузеров поддерживают Java, однако выполнение аплетов может быть заблокировано пользователем или системным администратором. Тогда для вывода на экран сообщения следует воспользоваться атрибутом ALT.

```
<applet code="CalculatorApplet.class" width=100 height=150
alt=" Если бы ваш браузер поддерживал язык Java, здесь вы увидели бы каль-
кулятор">
```

Дескриптор <object>

Дескриптор <object> определен в спецификации HTML 4.0. Он предложен консорциумом W3C взамен дескриптора <applet>. У этого дескриптора есть 35 атрибутов, большинство из которых (например, onkeydown) предназначено для разработчиков, использующих Dynamic HTML. Атрибуты, определяющие размеры и местоположение аплета, например align и height, ничем не отличаются от таких же атрибутов дескриптора applet. Ключевым атрибутом дескриптора object, имеющим отношение к Java-аплетам, является classid. Этот атрибут ссылается на объект аплета. Помимо аплетов, с помощью дескриптора <object> можно загружать и другие объекты, например компоненты ActiveX. Тип загружаемого объекта указывается посредством атрибута codetype. Аплет, написанный на языке Java, имеет тип application/java. Для его загрузки нужно задать в дескрипторе <object> следующие атрибуты:

```
<object
   codetype="application/java"
   classid="java:CalculatorApplet.class"
   width="100" height="150">
```

Обратите внимание на то, что атрибут classid может сопровождаться атрибутом codebase точно так же, как это происходит в дескрипторе <applet>.

Передача информации аплетам

Подобно тому, как приложения могут воспринимать информацию, указанную в командной строке, аплеты могут использовать параметры, заданные в HTML-файле. Эта задача решается с помощью HTML-дескриптора <param>. Допустим, что вам нужно определить в составе Web-страницы стиль шрифта, использованного в аплете. Для этого понадобится задать следующие дескрипторы:

```
<applet code="FontParamApplet.class" width=200, height=200>
<param name="font" value="Helvetica" />
</applet>
```

Аплет получает значение параметра с помощью метода getParameter(), определенного в классе Applet.

```
public class FontParamApplet extends JApplet
{
   public void init()
```

```
   {
      String fontName = getParameter("font");
      ...
   }
   ...
}
```

Метод `getParameter()` можно вызывать только из метода `init()`, но не из конструктора аплета. После выполнения конструктора параметры еще не готовы. Поскольку компоновка нетривиальных аплетов определяется параметрами, мы не рекомендуем использовать конструкторы. Весь код, необходимый для инициализации, следует поместить в метод `init()`.

Параметры всегда возвращаются как строки. Для того чтобы получить числовую информацию, необходимо преобразовать эти строки в числа. Это можно сделать, пользуясь стандартным способом, например, вызвав метод `parseInt()` класса `Integer`.

Допустим, мы хотим добавить параметр, задающий размер шрифта. Тогда соответствующий код на языке HTML будет выглядеть следующим образом:

```
<applet code="FontParamApplet.class" width="200" height="200">
   <param name="font" value="Helvetica"/>
   <param name="size" value="24"/>
</applet>
```

Считать этот целочисленный параметр можно с помощью приведенного ниже фрагмента программы.

```
public class FontParamApplet extends JApplet
{
   public void init()
   {
      String fontName = getParameter("font");
      int fontsize = Integer.parseInt(getParameter("size"));
      ...
   }
}
```

Строки, используемые для определения параметров в дескрипторе `<param>`, и строки в методе `getParameter` должны совпадать абсолютно точно. В частности, в них должны различаться прописные и строчные буквы.

Кроме того, для гарантии, что параметры заданы правильно, нужно проверить, *не пропущено ли значение параметра* `size`. *Для этого достаточно сравнить его* со значением `null`.

```
int fontsize;
String sizeString = getParameter('size');
if (sizeString == null) fontSize = 12;
else fontSize = Integer.parsent(sizeString);
```

Ниже приведен пример аплета, широко использующего параметры. Этот аплет рисует на экране гистограмму, показанную на рис. 10.7.

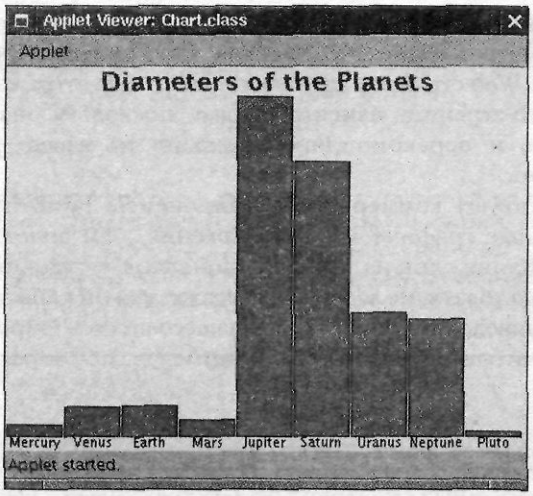


Рис. 10.7. Аплет, рисующий гистограмму

Аплет получает в качестве параметров метки и числа, определяющие высоту столбиков гистограммы. Требуемые значения задаются в HTML-файле с помощью дескриптора <param>. Ниже показано, как выглядит HTML-файл, соответствующий рис. 10.7.

```
<applet code="Chart.class" width="400" height="300">
   <param name="title" value="Diameters of the Planets"/>
   <param name="values" value="9"/>
   <param name="name.1" value="Mercury"/>
   <param name="name.2" value="Venus"/>
   <param name="name.3" value="Earth"/>
   <param name="name.4" value="Mars"/>
   <param name="name.5" value="Jupiter"/>
   <param name="name.6" value="Saturn"/>
   <param name="name.7" value="Uranus"/>
   <param name="name.8" value="Neptune"/>
   <param name="name.9" value="Pluto"/>
   <param name="value.1" value="3100"/>
   <param name="value.2" value="7500"/>
   <param name="value.3" value="8000"/>
   <param name="value.4" value="4200"/>
   <param name="value.5" value="88000"/>
   <param name="value.6" value="71000"/>
   <param name="value.7" value="32000"/>
   <param name="value.8" value="30600"/>
   <param name="value.9" value="1430"/>
</applet>
```

Эти значения можно было бы задать в виде массива строк или чисел, однако у дескрипторов `param` есть два преимущества. Во-первых, можно создать несколько копий одного и того же аплета на Web-странице, демонстрируя разные гистограммы: для этого достаточно поместить на страницу два дескриптора `applet` с разными наборами параметров. Во-вторых, данные, которые отображаются в виде гистограмм,

можно изменять. Приведенный выше пример не очень хорошо демонстрирует сказанное, так как диаметры планет не изменяются. Поэтому предположим, что нам нужно поместить на Web-страницу другую гистограмму, отражающую объем еженедельных продаж. Web-страницу изменить легко, поскольку она содержит обычный текст. Редактировать и перекомпилировать файл на языке Java каждую неделю довольно утомительно.

Разумеется, существуют коммерческие компоненты JavaBeans, которые создают намного более сложные графические изображения, чем аплет, приведенный здесь в качестве примера. Купив один из таких компонентов, можно поместить его на свою Web-страницу и использовать, не задумываясь даже, как он работает.

В листинге 10.5 приведен исходный текст нашего аплета. Обратите внимание на то, что метод `init()` считывает параметры, а метод `paintComponent()` рисует гистограмму.

Листинг 10.5. Содержимое файла `Chart.java`

```
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import javax.swing.*;

public class Chart extends JApplet
{
   public void init()
   {
      String v = getParameter("values");
      if (v == null) return;
      int n = Integer.parseInt(v);
      double[] values = new double[n];
      String[] names = new String[n];
      for (int i = 0; i < n; i++)
      {
         values[i] = Double.parseDouble(getParameter
            ("value." + (i + 1)));
         names[i] = getParameter("name." + (i + 1));
      }

      add(new ChartPanel(values, names, getParameter("title")));
   }
}

/**
    Панель с гистограммой.
*/
class ChartPanel extends JPanel
{
   /**
       Конструктор ChartPanel.
       @param v Массив значений гистограммы
       @param n Массив меток гистограммы
       @param t Заголовок гистограммы
   */
   public ChartPanel(double[] v, String[] n, String t)
   {
```

```
      values = v;
      names = n;
      title = t;
   }

   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;

      // Вычисление минимального и максимального значений.
      if (values == null) return;
      double minValue = 0;
      double maxValue = 0;
      for (double v : values)
      {
         if (minValue > v) minValue = v;
         if (maxValue < v) maxValue = v;
      }
      if (maxValue == minValue) return;

      int panelWidth = getWidth();
      int panelHeight = getHeight();

      Font titleFont = new Font("SansSerif", Font.BOLD, 20);
      Font labelFont = new Font("SansSerif", Font.PLAIN, 10);

      // Вычисление длины заголовка.
      FontRenderContext context = g2.getFontRenderContext();
      Rectangle2D titleBounds =
         titleFont.getStringBounds(title, context);
      double titleWidth = titleBounds.getWidth();
      double top = titleBounds.getHeight();

      // Отображение заголовка.
      double y = -titleBounds.getY();
      double x = (panelWidth - titleWidth) / 2;
      g2.setFont(titleFont);
      g2.drawString(title, (float) x, (float) y);

      // Вычисление размеров меток гистограммы.
      LineMetrics labelMetrics =
         labelFont.getLineMetrics("", context);
      double bottom = labelMetrics.getHeight();

      y = panelHeight - labelMetrics.getDescent();
      g2.setFont(labelFont);

      // Вычисление масштаба и ширины полос.
      double scale = (panelHeight - top - bottom) /
         (maxValue - minValue);
      int barWidth = panelWidth / values.length;

      // Отображение полос.
      for (int i = 0; i < values.length; i++)
      {
         // Получение координат прямоугольников.
```

```
            double x1 = i * barWidth + 1;
            double y1 = top;
            double height = values[i] * scale;
            if (values[i] >= 0)
               y1 += (maxValue - values[i]) * scale;
            else
            {
               y1 += maxValue * scale;
               height = -height;
            }

            // Заполнение полос и рисование контуров.
            Rectangle2D rect = new Rectangle2D.Double(x1, y1,
               barWidth - 2, height);
            g2.setPaint(Color.RED);
            g2.fill(rect);
            g2.setPaint(Color.BLACK);
            g2.draw(rect);

            // Рисование метки, выровненной по центру.
            Rectangle2D labelBounds = labelFont.getStringBounds
               (names[i], context);

            double labelWidth = labelBounds.getWidth();
            x = x1 + (barWidth - labelWidth) / 2;
            g2.drawString(names[i], (float) x, (float) y);
         }
      }

      private double[] values;
      private String[] names;
      private String title;
}
```

API **java.applet.Applet 1.0**

- `public String getParameter(String name)`

  Получает параметры, которые задаются посредством дескриптора <param>, расположенного на Web-странице. Прописные и строчные буквы различаются.

- `public String getAppletInfo()`

  Возвращает строку, содержащую информацию об авторе, версии программы и авторских правах на данный аплет. Требуемая информация задается при переопределении метода.

- `public String[][] getParameterInfo()`

  Данный метод переопределяется так, чтобы он возвращал массив параметров, поддерживаемых данным аплетом. В каждой строке содержится три элемента: имя, тип и описание параметра. Ниже приведен пример информации, возвращаемой в результате выполнения данного метода.

```
"fps", "1-10", "frames per second"
"repeat", "boolean", "repeat image loop?"
"images", "url", "directory containing images"
```

Мультимедиа

Аплеты могут отображать на экране графические изображения и воспроизводить аудиофайлы. Как известно, графические изображения представляются в форматах GIF или JPEG, а аудиофайлы — в форматах AU, AIFF, WAV или MIDI. Существует возможность создания анимационных GIF-изображений. Обычно ссылка на анимационные данные представляет собой URL файла.

Инкапсуляция URL

URL представляют собой описания ресурсов глобальной сети. Например, адрес `http://java.sun.com/index.html` сообщает браузеру о том, что файл `index.html` расположен на сервере `java.sun.com` и при обращении к нему нужно применить протокол HTTP (HyperText Tranfer Protocol — протокол передачи гипертекстовых данных). В языке Java есть класс `URL`, инкапсулирующий URL. Простейший способ создать URL — задать строку в конструкторе класса `URL`.

```
URL u = new URL("http://java.sun.com/index.html");
```

Такой URL-адрес называется *абсолютным*, так как мы указали полное имя ресурса. Другой полезный конструктор класса `URL` позволяет создавать *относительные* URL.

```
URL data = new URL(u, "data/planets.dat");
```

Этот конструктор задает относительный путь к файлу `planet.dat`, расположенному в каталоге `data`. Путь вычисляется относительно базового адреса `u`.

При работе с аплетами требуются ответы на два вопроса.

- Какой URL имеет страница, содержащая аплет?
- Какой URL имеет сам аплет?

Ответ на первый вопрос дает метод `getDocumentBase()`, а на второй — метод `getCodeBase()`.

В предыдущих версиях JDK при описании обоих методов была допущена ошибка (зарегистрированная под номером #4456393). Эта ошибка была исправлена в JDK 5.0.

Используя Java Plug-In, можно получать из аплетов доступ к защищенным Web-страницам (подробно данный вопрос рассмотрен в документе `http://java.sun.com/products/plugin/1.3/docs/https.html`). При этом взаимодействие осуществляется по протоколу SSL.

Получение мультимедиа-файлов

Файлы с графическими изображениями и аудиоданные можно получать с помощью методов `getImage()` и `getAudioClip()`.

```
Image cat = getImage(getCodeBase(), "images/cat.gif");
AudioClip meow = getAudioClip(getCodeBase(), "audio/meow.au");
```

Здесь мы применили метод `getCodeBase()`, возвращающий URL аплета. Второй параметр метода задает расположение графических изображений и аудиозаписей относительно базового документа.

Изображения и аудиофайлы должны размещаться на том же сервере, что и аплет. По соображениям безопасности аплеты не имеют доступа к файлам, расположенным на других серверах (принцип "аплеты могут звонить только домой").

В главе 7 мы показали, как на экран выводится изображение. Для того чтобы воспроизвести аудиозапись, достаточно вызвать метод `play()`.

```
play(getCodeBase(), "audio/meow.au");
```

Для ускорения загрузки мультимедийные объекты можно размещать в JAR-файлах (JAR-файлы будут рассмотрены далее в этой главе). Методы `getImage()`, `getAudioClip` и `play` автоматически распознают архивы. Если графическое изображение или аудиозапись содержатся в JAR-файле, они немедленно загружаются. В противном случае браузер запрашивает их с Web-сервера.

java.net.URL 1.0

- `URL(String name)`

  Создает объект `URL`, используя строку, описывающую абсолютный URL.

- `URL(URL base, String name)`

  Создает относительный объект, описывающий относительный URL. Если строка `name` представляет собой абсолютный URL, то параметр `base` игнорируется. В противном случае `base` принимается в качестве базового адреса.

java.applet.Applet 1.0

- `URL getDocumentBase()`

  Возвращает URL страницы, содержащей аплет.

- `URL getCodeBase()`

  Возвращает URL каталога, содержащего аплет. Это может быть либо абсолютный URL, либо ссылка на каталог, указываемый с помощью атрибута `codebase`, либо, если данный атрибут отсутствует, ссылка на каталог, в котором расположен HTML-файл.

- `void play(URL url)`
- `void play(URL url, String name)`

  Первый метод воспроизводит аудиофайлы, расположенные по указанному URL, а второй интерпретирует строку как адрес, определяемый относительно URL, заданного первым параметром. Если аудиофайл не найден, никакие действия не выполняются.

- `AudioClip getAudioClip(URL url)`
- `AudioClip getAudioClip(URL url, String name)`

  Первый метод получает аудиозапись, расположенную по заданному URL-адресу, а второй интерпретирует строку как адрес, определяемый относительно URL, заданного первым параметром. Если аудиофайл не найден, оба метода возвращают значение `null`.

- `Image getImage(URL url)`
- `Image getImage(URL url, String name)`

  Возвращает *объект класса* `Image`, *инкапсулирующий графическое изображение*, заданное своим URL. Если изображения по указанному адресу нет, метод немедленно возвращает `null`. В противном случае запускается отдельный поток для загрузки изображения.

Контекст аплета

Аплет запускается либо браузером, либо программой `appletviewer`. Аплет может поручить браузеру выполнить некоторые действия, например, извлечь аудиоданные, отобразить короткое сообщение в строке состояния или отобразить другую Web-страницу. Браузер, содержащий аплет, может удовлетворить его запросы либо проигнорировать их. Например, если аплет выполняется в среде программы `appletviewer`, его требование отобразить Web-страницу не будет удовлетворено.

Для взаимодействия с браузером аплет вызывает метод `getAppletContext()`. Этот метод возвращает объект, реализующий интерфейс `AppletContext`. Конкретную *реализацию интерфейса* `AppletContext` *можно считать средством взаимодействия* аплета и содержащего его браузера. В дополнение к методам `getAudioClip()` и `getImage()` в интерфейсе `AppletContext` объявлено несколько полезных методов, которые обсуждаются ниже.

Взаимодействие аплетов

Web-страница может содержать несколько аплетов. Если параметр `codebase` у этих аплетов один и тот же, они могут взаимодействовать друг с другом. Естественно, что взаимодействие аплетов — довольно сложный процесс, необходимость в котором возникает не часто.

Если для каждого аплета задать атрибут `name`, то с помощью метода `getApplet(String)`, объявленного в интерфейсе `AppletContext`, можно получить ссылку на любой аплет. Предположим, что в HTML-файле содержится дескриптор

```
<applet code="Chart.class" width=100 height=100 name="Chart1">
```

В этом случае в результате вызова приведенного ниже метода будет получена ссылка на аплет.

```
Applet chart1 = getAppletContext().getApplet("Chart1");
```

Что можно сделать с этой ссылкой? Если в классе `Chart` есть метод, предназначенный для получения новых данных и обновления гистограммы, его можно вызвать, выполнив соответствующее приведение типов.

```
((Chart)chart1).setData(3, "Earth", 9000);
```

Кроме того, можно перечислить все аплеты, размещенные на Web-странице, независимо от того, имеют они атрибут `name` или нет. Метод `getApplets()` возвращает объект `Enumeration` (он будет подробно рассмотрен во втором томе). Этот объект, *в частности, предоставляет доступ ко всем аплетам. Ниже приведен фрагмент кода*, выводящий имена аплетов, расположенных на текущей Web-странице.

```
Enumeration e = getAppletContext().getApplets();
while (e.hasMoreElements())
{
   Object a = e.nextElements();
   System.out.println(a.getClass().getName());
}
```

Аплет не может взаимодействовать с аплетами, находящимися на другой Web-странице.

Отображение элементов в браузере

У программиста есть доступ к двум областям браузера, содержащего аплет: строке состояния и области отображения Web-страницы. В обоих случаях используются методы класса `AppletContext`.

С помощью метода `showStatus()` можно вывести строку сообщения в строке состояния.

```
showStatus("Загрузка данных ... пожалуйста, ждите");
```

Опыт показывает, что метод `showStatus()` имеет ограниченную область применения. Браузер также использует строку состояния и в большинстве случаев заменяет ранее выведенную строку своими сообщениями, например `Applet running`. Не используйте строку состояния для вывода важных сообщений, которые пользователь не должен пропустить.

С помощью метода `showDocument()` можно отобразить другую Web-страницу. Простейший способ: вызвать метод `showDocument()` с одним параметром, задающим URL нужного документа.

```
URL u = new URL("http://java.sun.com/index.html");
getAppletContext().showDocument(u);
```

При этом возникает одна проблема. Дело в том, что браузер открывает новую Web-страницу в том же самом окне, в котором отображался текущий документ, заменяя таким образом ваш аплет. Чтобы снова получить возможность работать с аплетом, пользователь должен щелкнуть на кнопке Back, расположенной на панели инструментов браузера.

Вызывая метод `showDocument()` с двумя параметрами, можно указать браузеру на то, что документ должен быть выведен в другом окне. Значения второго параметра описаны в табл. 10.2. Второй параметр является строкой. Если вы зададите специальную строку `"_blank"`, браузер откроет новое окно, не изменяя текущее. Что еще важнее, можно разделить окно браузера на несколько фреймов каждое, и присвоить каждому из них имя. Затем можно поместить аплет внутрь фрейма, а остальные документы отображать в других фреймах.

Программа `appletviewer` не отображает Web-страницы, поэтому она игнорирует метод `showDocument()`.

Таблица 10.2. Значения второго параметра метода showDocument()

Значение параметра	Место вывода документа
`"_self"` или отсутствует	Текущий фрейм
`"_parent"`	Родительский фрейм
`"_top"`	Фрейм самого высокого уровня
`"_blank"`	Новое безымянное окно верхнего уровня
Любая другая строка	Фрейм, имеющий заданное имя. Если фрейма с таким именем не существует, открывается новое окно, получающее заданное имя

java.applet.Applet 1.2

- `public AppletContext getAppletContext()`

  Предоставляет описание среды, в которой выполняется аплет. В большинстве случаев эта информация используется для управления браузером.

- `void showStatus(String msg)`

  Выводит в строке состояния браузера указанную строку.

java.applet.AppletContext 1.2

- `Enumeration getApplets()`

  Возвращает объект, содержащий ссылки на все аплеты, находящиеся в одном и том же контексте, т.е. на одной и той же Web-странице.

- `Applet getApplet(String name)`

  Возвращает аплет, находящийся в текущем контексте и имеющий заданное имя. Если такого аплета не существует, возвращает ссылку `null`. Поиск производится только на текущей Web-странице.

- `void showDocument(URL url)`
- `void showDocument(URL url, String target)`

  Отображает новую Web-страницу в фрейме. При вызове первого метода новая страница заменяет собой текущую. При вызове второго метода используется строка, идентифицирующая нужный фрейм.

Аплет для поддержки закладок

Данный аплет позволяет воспользоваться преимуществами фреймов. Разделим экран вертикально на два фрейма. Левый фрейм содержит аплет на языке Java, отображающий список закладок. Когда пользователь выбирает одну из них, соответствующая Web-страница отображается в правом фрейме (рис. 10.8).

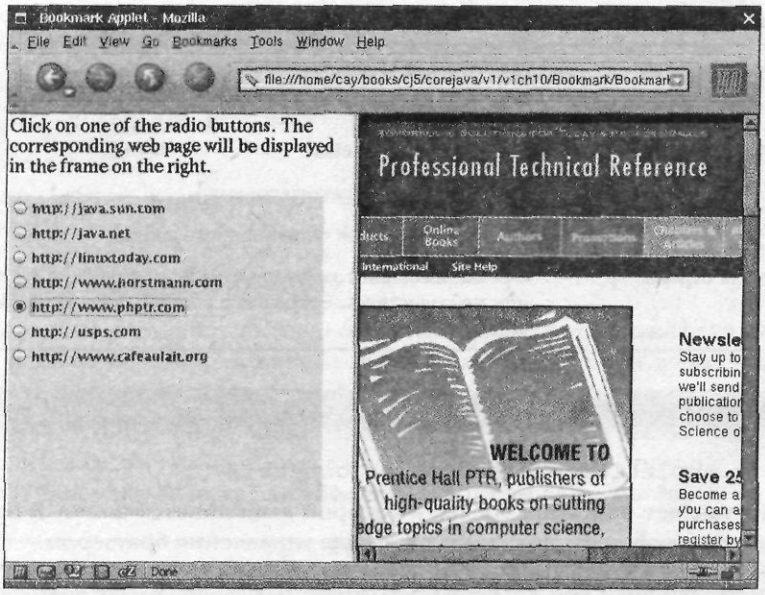


Рис. 10.8. Аплет, поддерживающий закладки

В листинге 10.6 показан HTML-файл, определяющий фреймы.

Листинг 10.6. Содержимое файла `Bookmark.html`

```
<html>
   <head>
      <title>Bookmark Applet</title>
   </head>
   <frameset cols="320,*">
      <frame name="left" src="Left.html"
         marginheight="2" marginwidth="2"
         scrolling="no" noresize="noresize"/>
      <frame name="right" src="Right.html"
         marginheight="2" marginwidth="2"
         scrolling="yes" noresize="noresize"/>
   </frameset>
</html>
```

Мы не будем подробно рассматривать синтаксические конструкции, использованные в данном документе. Следует лишь отметить, что каждый фрейм обладает двумя важными свойствами: именем (заданным атрибутом `name`) и URL (заданным атрибутом `src`). Мы не придумали ничего лучше, как назвать фреймы `left` и `right`.

Левый фрейм (листинг 10.7) загружает файл `Left.html`, который в свою очередь загружает в него аплет. В данном файле указаны лишь аплет и закладки.

Листинг 10.7. Содержимое файла `Left.html`

```
<html>
   <head><title>A Bookmark Applet</title></head>
   <body>
```

```
      <p>
         Щелкните на одном из переключателей.
         Соответствующая страница отобразится в правом фрейме.
      </p>
      <applet code="Bookmark.class" width="290" height="300">
         <param name="link.1" value="http://java.sun.com"/>
         <param name="link.2" value="http://java.net"/>
         <param name="link.3" value="http://linuxtoday.com"/>
         <param name="link.4" value="http://www.horstmann.com"/>
         <param name="link.5" value="http://www.phptr.com"/>
         <param name="link.6" value="http://usps.com"/>
         <param name="link.7" value="http://www.cafeaulait.org"/>
      </applet>
   </body>
</html>
```

В правом фрейме сначала отображается специально подготовленный файл `Right.html` (листинг 10.8). Дело в том, что пустые фреймы оставлять не рекомендуется. Желательно выводить в них какие-нибудь данные.

Листинг 10.8. Содержимое файла `Right.html`

```
<html>
   <head>
   <title>Здесь будет отображаться Web-страница.</title>
   </head>
   <body>
      <p>Щелкните на переключателе в левом фрейме.
      Здесь отобразится соответствующая Web-страница. </p>
   </body>
</html>
```

Код аплета, приведенный в листинге 10.9, довольно прост. Он считывает значения параметров `link_1`, `link_2` и т.д. и превращает их в переключатели. При выборе переключателя метод `showDocument()` отображает в правом фрейме соответствующую страницу.

Листинг 10.9. Содержимое файла `Bookmark.java`

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.*;
import java.net.*;
import javax.swing.*;

public class Bookmark extends JApplet
{
   public void init()
   {
      Box box = Box.createVerticalBox();
      ButtonGroup group = new ButtonGroup();

      int i = 1;
      String urlString;
```

```
      // Чтение параметров link.n.
      while ((urlString = getParameter("link." + i)) != null)
      {

         try
         {
            final URL url = new URL(urlString);

            // Создание переключателя для каждой ссылки.
            JRadioButton button = new JRadioButton(urlString);
            box.add(button);
            group.add(button);

            // Выбор переключателя приводит к отображению
            // в правом фрейме соответствующего документа.
            button.addActionListener(new
               ActionListener()
               {
                  public void actionPerformed(ActionEvent event)
                  {
                     AppletContext context = getAppletContext();
                     context.showDocument(url, "right");
                  }
               });
         }
         catch (MalformedURLException e)
         {
            e.printStackTrace();
         }

         i++;
      }

      add(box);
   }
}
```

Аплет или приложение? И то и другое!

Несколько лет назад юмористическая программа "Субботний вечер в прямом эфире" на американском коммерческом телевизионном канале показала спор супругов о белой желеобразной субстанции. Муж сказал: "Это – десертное блюдо". Жена сказала: "Это – мастика для паркета". А ведущий триумфально подвел итог: "Это – и то и другое!"

В этом разделе мы покажем, как писать программы, представляющие собой и аплет, и приложение одновременно. Такую программу можно выполнять с помощью `appletviewer` или в среде браузера либо запускать из командной строки под управлением интерпретатора `java`. Мы не уверены, что это так уж необходимо, – просто интересно, что подобная задача оказалась в принципе разрешимой.

На рис. 10.9 и 10.10 показана *одна и та же программа*, запущенная из командной строки как приложение и в среде `appletviewer` как аплет.

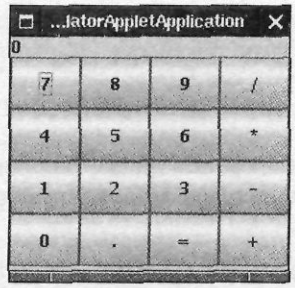


Рис. 10.9. Калькулятор как приложение

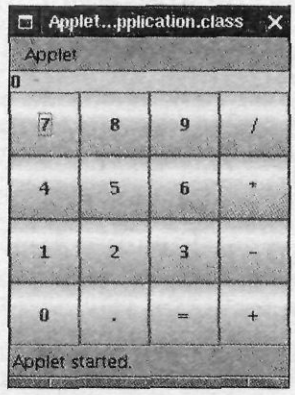


Рис. 10.10. Калькулятор как аплет

Каждый файл класса содержит только один общедоступный класс. Чтобы запустить аплет, этот класс должен быть подклассом класса `Applet`. Чтобы запустить приложение, нужен статический метод `main()`. Итак, нужно написать код, подобный приведенному ниже.

```
class MyAppletApplication extends JApplet
{
   public void init() { ... }
   ...
   public static void main(String[] args) { ... }
}
```

Как должен выглядеть метод `main()`? Обычно в нем создается экземпляр класса и вызывается метод `setVisible(true)`. Однако в нашем случае задача не так проста. Метод `setVisible()` для обычного аплета вызывать нельзя. Аплет должен находиться в составе фрейма.

Для того чтобы сформировать фрейм, создадим класс `AppletFrame`.

```
public class AppletFrame extends JFrame
{
   public AppletFrame(Applet anApplet)
   {
      applet = anApplet;
```

```
      add(applet);
      ...
   }
   ...
}
```

Конструктор фрейма помещает аплет (расширяющий класс `Component`) на панель содержимого. В методе `main()` нашего аплета/приложения создадим и отобразим `AppletFrame`.

```
class MyAppletApplication extends JApplet
{
   public void init() { . . . }
   . . .
   public static void main(String args[])
   {
      AppletFrame frame = new AppletFrame(new
         MyAppletApplication());
      frame.setTitle("MyAppletApplication");
      frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

При запуске аплета должны вызываться методы `init()` и `start()`. Мы обеспечим их запуск, переопределив метод `setVisible()`.

```
public void setVisible(boolean b)
{
   if (b)
   {
      applet.init();
      super.setVisible(true);
      applet.start();
   }
   else
   {
      applet.stop();
      super.setVisible(false);
      applet.destroy();
   }
}
```

Однако здесь таится ловушка. Если программа запускается интерпретатором, а не программой `appletviewer`, и вызывает метод `getAppletContext()`, она получит в ответ ссылку `null`, поскольку запускается вне среды браузера. Во время работы программы попытка выполнить приведенное ниже выражение закончится неудачей.

```
getAppletContext().showStatus(message);
```

Мы не собираемся писать полнофункциональный браузер, достаточно правильно обработать указанный вызов. Он не выводит на экран никаких сообщений, но, по крайней мере, не приводит к аварийному завершению программы. Для того чтобы правильно его выполнить, достаточно реализовать два интерфейса: `AppletStub` и `AppletContext`.

Вы уже видели контексты аплетов в действии. Они выполняют загрузку графических и аудиофайлов, а также отображают на экране Web-страницы. Однако контексты могут вежливо отказаться выполнять команды, и именно это сделал контекст нашего аплета. Основная цель интерфейса `AppletStub` — обнаружить контекст аплета.

В нашем случае класс `AppletFrame` реализует интерфейсы `AppletStub` и `AppletContext` одновременно. Мы предусмотрели минимум функциональных возможностей, необходимых для реализации этих интерфейсов.

```
public class AppletFrame extends JFrame
   implements AppletStub, AppletContext
{
   . . .
   // Методы, объявленные в интерфейсе AppletStub
   public boolean isActive() { return true; }
   public URL getDocumentBase() { return null; }
   public URL getCodeBase() { return null; }
   public String getParameter(String name) { return ""; }
   public AppletContext getAppletContext() { return this; }
   public void appletResize(int width, int height) {}

   // Методы, объявленные в интерфейсе AppletContext
   public AudioClip getAudioClip(URL url) { return null; }
   public Image getImage(URL url) { return null; }
   public Applet getApplet(String name) { return null; }
   public Enumeration getApplets() { return null; }
   public void showDocument(URL url) {}
   public void showDocument(URL url, String target) {}
   public void showStatus(String status) {}
   public void setStream(String key, InputStream stream) {}
   public InputStream getStream(String key) { return null; }
   public Iterator getStreamKeys() { return null; }
}
```

Пытаясь скомпилировать этот файл с помощью компилятора, содержащегося в JDK 1.3, вы получите предупреждение, что класс `java.awt.Window` уже содержит метод `isActive()` с областью видимости, ограниченной пакетом. Поскольку наш класс и класс `Window` находятся в разных пакетах, метод `Window.isActive()` переопределить невозможно. Это нам подходит — ведь мы хотим создать новый метод `isActive()` для интерфейса `AppletStub`. Интересно, что в подкласс можно добавить новый метод с сигнатурой уже существующего. Как только программа получит доступ к объекту через ссылку `Window` в пакете `java.awt`, будет вызван метод `Window.isActive()`, область видимости которого ограничена пакетом. Однако если доступ к объекту обеспечивается через класс `AppletFrame` или интерфейс `AppletStub`, вызывается метод `AppletFrame.isActive()`.

Затем конструктор фрейма вызывает метод `setStub()` для аплета и создается его заглушка.

```
public AppletFrame(Applet anApplet)
{
   applet = anApplet;
   Container contentPane = getContentPane();
   contentPane.add(applet);
   applet.setStub(this);
}
```

И наконец, возможен еще один прием. Допустим, что мы хотим использовать калькулятор в качестве аплета и приложения одновременно. Вместо того чтобы перемещать методы класса `CalculatorApplet` в класс `CalculatorAppletApplication`, мы используем обычное наследование. Ниже приведен пример класса.

```
public class CalculatorAppletApplication extends CalculatorApplet
{
   public static void main(String args[])
   {
   AppletFrame frame
      = new AppletFrame(new CalculatorApplet());
   ...
   }
}
```

Такое можно проделать с любым аплетом, а не только с калькулятором. Для этого нужно лишь сделать класс `MyAppletApplication` подклассом аплета и передать объект `new MyApplet()` методу `AppletFrame` в методе `main()`. В результате вы получите класс, представляющий собой и аплет, и приложение одновременно.

Хотя это и необязательно, мы оформили дескриптор `<applet>` в виде комментариев. Теперь можно вызвать программу `appletviewer`, задав в командной строке имя исходного (!) файла и не прибегая к помощи HTML-файлов.

Соответствующий код приведен в листингах 10.10 и 10.11. Чтобы скомпилировать эту программу, нужно скопировать файл `CalculatorApplet.java` в ее каталог. Запустить аплет/приложение можно с помощью следующих команд:

```
appletviewer CalculatorAppletApplication.java
java CalculatorAppletApplication
```

Листинг 10.10. Содержимое файла `AppletFrame.java`

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;

public class AppletFrame extends JFrame
   implements AppletStub, AppletContext
{
   public AppletFrame(Applet anApplet)
   {
      applet = anApplet;
      add(applet);
      applet.setStub(this);
   }

   public void setVisible(boolean b)
   {
      if (b)
      {
         applet.init();
         super.setVisible(true);
```

```
            applet.start();
         }
         else
         {
            applet.stop();
            super.setVisible(false);
            applet.destroy();
         }
      }

      // Методы, объявленные в интерфейсе AppletStub
      public boolean isActive() { return true; }
      public URL getDocumentBase() { return null; }
      public URL getCodeBase() { return null; }
      public String getParameter(String name) { return ""; }
      public AppletContext getAppletContext() { return this; }
      public void appletResize(int width, int height) {}

      // Методы, объявленные в интерфейсе AppletContext
      public AudioClip getAudioClip(URL url) { return null; }
      public Image getImage(URL url) { return null; }
      public Applet getApplet(String name) { return null; }
      public Enumeration getApplets() { return null; }
      public void showDocument(URL url) {}
      public void showDocument(URL url, String target) {}
      public void showStatus(String status) {}
      public void setStream(String key, InputStream stream) {}
      public InputStream getStream(String key) { return null; }
      public Iterator getStreamKeys() { return null; }

      private Applet applet;
}
```

Листинг 10.11. Содержимое файла `CalculatorAppletApplication.java`

```
/*
  Если для выполнения аплета используется программа
  appletviewer, то она считывает указанные ниже дескрипторы.

  appletviewer CalculatorAppletApplication.java (!)

  В этом случае отдельный HTML-файл оказывается ненужным.

  <applet code="CalculatorAppletApplication.class"
   width="200" height="200">
  </applet>
*/

import javax.swing.*;

public class CalculatorAppletApplication
   extends CalculatorApplet
// Программа, являющаяся одновременно и аплетом, и приложением.
{
   public static void main(String[] args)
   {
```

```
        AppletFrame frame = new AppletFrame(new CalculatorApplet());
        frame.setTitle("CalculatorAppletApplication");
        frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    public static final int DEFAULT_WIDTH = 200;
    public static final int DEFAULT_HEIGHT = 200;
}
```

Файлы JAR

Аплет, рассматриваемый в данной главе и реализующий функции калькулятора, использует четыре класса: `CalculatorApplet`, `CalculatorPanel` и два внутренних класса. Дескриптор аплета ссылается на файл класса, представляющего собой подкласс `JApplet`:

```
<applet code="CalculatorApplet.class" width=100 height=150>
```

Прочитав этот файл, браузер соединится с Web-сервером и загрузит файл `CalculatorApplet.class`. Затем *загрузчик классов* (class loader) виртуальной машины Java, встроенной в браузер, извлечет класс `CalculatorApplet` из этого файла. В это время загрузчик должен выявить другие классы, используемые в загружаемом классе. Сделав это, загрузчик обнаружит, что для запуска аплета нужны дополнительные классы, и тогда браузер вновь соединится с Web-сервером. Большинство аплетов состоит из нескольких классов, и браузер вынужден устанавливать много соединений с Web-сервером, копируя каждый файл класса отдельно. Загрузка такого аплета может происходить довольно медленно.

Следует помнить, что загрузка выполняется так медленно не из-за большого размера файлов, содержащих классы, — они достаточно малы. Причина заключается в излишних затратах времени на соединение с Web-сервером.

Чтобы оптимизировать процесс загрузки, можно упаковать все необходимые классы в один файл. Затем этот файл копируется с помощью *единственного* HTTP-запроса. Файлы, в которых архивируются файлы классов, называются JAR-архивами. JAR-файлы могут содержать не только файлы классов, но и файлы других типов, например графические изображения и аудиофайлы. JAR-файлы сжимаются с помощью хорошо известного формата ZIP, сокращая время их загрузки.

Для создания JAR-файлов используется инструмент `jar`. (Как правило, утилита `jar` располагается в каталоге `jdk/bin`.) Новый JAR-файл создается с помощью выражения, подобного следующему:

```
jar cvf имя_JAR-файла файл_1 файл_2 ...
```

Пример вызова программы `jar` приведен ниже

```
jar cvf CalculatorClasses.jar *.java icon.gif
```

Общий формат команды `jar` выглядит следующим образом:

```
jar опции файл_1 файл_2 ...
```

В табл. 10.3 перечислены все опции программы `jar`. Они напоминают опции программы `tar`, хорошо известной пользователям операционной системе Unix.

Таблица 10.3. Опции программы `jar`

Опция	Описание
`c`	Создает новый или пустой архив и добавляет в него файлы. Если в качестве имени файла указано имя каталога, программа `jar` обрабатывает его рекурсивно
`t`	Отображает на экране содержимое архива
`u`	Обновляет существующий JAR-файл
`x`	Извлекает файлы. Если указано несколько имен файлов, извлекаются только они. В противном случае извлекаются все файлы
`f`	Задает имя JAR-файла в качестве второго параметра командной строки. Если этот параметр пропущен, то программа `jar` запишет результат в стандартный поток вывода (при создании JAR-файла) или считает его из стандартного потока ввода (при извлечении или просмотре содержимого JAR-файла)
`v`	Выводит подробные данные
`m`	Добавляет в JAR-архив *файл манифеста* (manifest), представляющий собой описание содержимого архива и его происхождения. Файл манифеста создается по умолчанию для каждого архива, но для подробного описания содержимого JAR-файла можно создать свой собственный манифест. Эта тема подробно обсуждается во втором томе
`0`	Помещает данные в архив, не упаковывая их в формате ZIP
`M`	Отменяет создание манифеста
`i`	Создает индексный файл
`C`	Временно изменяет каталог. Например, команда `jar cvf JARFileName.jar -C classes *.class` направляет файлы в подкаталог `classes`

Если у нас есть JAR-файл, на него нужно ссылаться в дескрипторе `applet`. Сделать это можно следующим образом:

```
<applet code="CalculatorApplet.class"
   archive="CalculatorClasses.jar"
   width=100 height=150>
```

Учтите, что атрибут `code` необходим. Он сообщает браузеру имя аплета. Атрибут `archive` задает имя файла, в котором находится класс аплета и другие файлы. Если в программе нужны файлы, содержащие классы, графические изображения или аудиозаписи, сначала браузер ищет JAR-файлы в списке, указанном с помощью параметра `archive`. Только если этих файлов там нет, браузер соединяется с Web-сервером.

Если ваш аплет слишком велик, не все смогут полностью использовать все его функциональные возможности. Для того чтобы сократить время загрузки, можно разбить код аплета, записав части в несколько JAR-файлов и добавив в главный JAR-файл *индексные данные*. Затем загрузчик классов распознает, какой JAR-файл содержит конкретный пакет или ресурс. Подробное описание процедуры индексирования содержится на Web-странице, расположенной по адресу `http://java.sun.com/j2se/5.0/docs/guide/jar/jar.html#JAR%20Index`.

В JDK 5.0 реализована новая схема сжатия, которая называется pack200. Она позволяет более эффективно упаковывать файлы классов по сравнению с традиционным алгоритмом сжатия ZIP. Специалисты компании Sun утверждают, что для больших JAR-файлов объемом в несколько мегабайт, содержащих только файлы классов, степень сжатия достигает 90%. Для доставки подобных архивов надо сконфигурировать специальным образом Web-сервер. Обычным клиентам он должен передавать стандартные файлы, а новым клиентам, сообщающим в HTTP-запросе о том, что они поддерживают алгоритм распаковки, — запакованные файлы. Более подробную информацию по этому вопросу можно найти по адресу `http://java.sun.com/j2se/5.0/docs/guide/deployment/deployment-guide/pack200.html`.

Java Plug-in может кэшировать коды аплетов. Подробно о настройке системы кэширования сказано в документе `http://java.sun.com/j2se/5.0/docs/guide/plugin/developer_guide/applet_caching.html`.

Упаковка приложений

Покинем мир аплетов и займемся упаковкой приложений, написанных на языке Java. Пересылая приложения, никто не хочет возиться с большим количеством файлов. Как и при использовании аплетов, нужно запаковать все классы и другие ресурсы, необходимые вашей программе, в JAR-файл. Запакованную программу можно загрузить с помощью простой команды или двойным щелчком на пиктограмме JAR-файла, *если операционная система настроена соответствующим образом.*

Файл манифеста

В JAR-файлы можно упаковывать приложения, компоненты и библиотеки программ. Например, *библиотека* JDK *содержится в очень большом архиве* `rt.jar`.

JAR-файл представляет собой обычный ZIP-архив, содержащий классы, другие файлы, необходимые для программы (например, пиктограммы), и *файл манифеста* (manifest file), описывающий особенности архива.

Файл манифеста называется `MANIFEST.MF` и находится в специальном подкаталоге `META-INF`. Минимально допустимый манифест не очень интересен:

```
Manifest-Version: 1.0
```

Сложные манифесты содержат гораздо более объемные описания, которые группируются в разделы. Первый раздел манифеста называется *главным*. Он относится ко всему JAR-файлу. Остальные описания относятся к отдельным файлам, пакетам или URL. Эти записи должны начинаться словом `Name`. Разделы отделяются друг от друга пустыми строками.

```
Manifest-Vewsrion: 1.0
Описание архива.

Name: Woozle.class
Описание файла.

Name: com/mycompany/mypkg/
Описание пакета.
```

Для того чтобы отредактировать этот манифест, поместите в него нужные строки. Затем выполните команду

```
jar cfm имя_JAR_файла имя_файла_манифеста
```

Например, для того чтобы создать новый JAR-файл с манифестом, вызовите утилиту jar следующим образом:

```
jar cfm MyArchive.jar manifest.mf com/mycompany/mypkg/*.class
```

Чтобы добавить в манифест существующего JAR-файла новые записи, запишите эти строки в текстовый файл, а затем выполните команду

```
jar ufm MyArchive.jar manfest-additions.mf
```

Более подробную информацию о форматах JAR-файлов и файлов манифеста можно найти на Web-странице `http://java.sun.com/j2se/5.0/docs/guide/jar/jar.html`.

Самораспаковывающиеся JAR-файлы

Для того чтобы запаковать приложение, надо поместить все файлы, необходимые для его работы, в JAR-архив и добавить запись манифеста, определяющую главный класс программы – тот, который нужно указать интерпретатору `java`.

Создайте файл `mainclass.mf`, содержащий строку

```
Main-Class: com/mycompany/mypkg/MainAppClass
```

Последняя строка манифеста должна заканчиваться символом перевода строки. В противном случае манифест не будет правильно считан. Данную ошибку разработчики допускают довольно часто.

Не указывайте расширения `.class` после имени главного класса. Затем выполните команду `jar`.

```
jar cvfm MyProgram.jar mainclass.mf добавляемые файлы
```

Теперь пользователь сможет просто запустить программу из командной строки.

```
java -jar MyProgram.jar
```

Если операционная система настроена соответствующим образом, приложение можно запустить, дважды щелкнув на пиктограмме JAR-файла.

- На платформе Windows система поддержки выполнения программ, написанных на языке Java, связывает расширение файла `.jar` с командой `javaw -jar`, которая запускает приложение. (В отличие от команды `java`, команда `javaw` не открывает окно для ввода команд оболочки.)
- На платформе Solaris операционная система распознает "волшебный номер" JAR-файла и запускает приложение с помощью команды `java -jar`.
- На платформе Mac OS X операционная система распознает расширение файла `.jar` и в ответ на двойной щелчок на соответствующей пиктограмме выполняет программу, написанную на языке Java. Более того, существует утилита

MRJAppBuilder, которую можно использовать для превращения JAR-файла в приложение операционной системы Mac. Детали изложены на Web-странице http://developer.apple.com/qa/java/java29.html.

Ресурсы

Классы, используемые в аплетах и приложениях, часто связаны с файлами, содержащими разнообразные данные. Примеры таких файлов перечислены ниже

- Графические и аудиофайлы.
- Текстовые файлы, содержащие строки сообщений и метки кнопок.
- Файлы, содержащие двоичные данные.

В языке Java все эти файлы называются *ресурсами* (resources).

В операционной системе Windows слово "ресурсы" имеет более узкий смысл. В этой системе к ресурсам также относятся пиктограммы, метки кнопок и т.п., но они должны быть связаны с исполняемыми файлами, а доступ к ним должен обеспечивать стандартный программный интерфейс. В противоположность этому ресурсы в языке Java находятся в отдельных файлах, а не в файлах, содержащих классы, и доступ к ним имеет любой класс.

Рассмотрим класс AboutPanel, выводящий на экран сообщение (рис. 10.11).

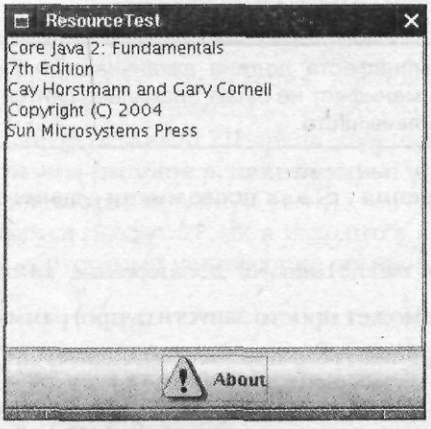


Рис. 10.11. Отображение ресурса из JAR-файла

Конечно, название книги и год выпуска, указанные на панели, в следующем издании книги изменятся. Для того чтобы легко отслеживать такие изменения, текст нужно поместить в отдельный файл, а не встраивать в программу в качестве строки.

Но где же следует разместить файл about.txt? Разумеется, было бы удобно поместить его вместе с остальными файлами программы, например в JAR-файле.

Загрузчик классов знает, где находятся файлы классов, если они содержатся в каталогах, указанных в переменной окружения CLASSPATH или в архиве или на Web-сервере. Механизм ресурсов дает программисту такие же удобства при работе с фай-

лами, которые не являются файлами классов. Для этого нужно выполнить следующие действия.

1. Получить объект `Class`, обладающий ресурсом, например `AboutPanel.class`.
2. Вызвать метод `getResource(filename)` и определить URL ресурса.
3. Если ресурс представляет собой графический или звуковой файл, прочитать его непосредственно с помощью методов `getImage()` или `getAudioClip()`.
4. В противном случае применить метод `openStream()` к URL и прочитать данные из файла (см. главу 12).

Дело в том, что загрузчик класса помнит, где располагается класс, и может обнаружить там же файлы ресурсов.

Например, чтобы создать пиктограмму, хранящуюся в файле `about.gif`, нужно использовать следующий фрагмент кода:

```
URL url = AboutPanel.class.getResource("about.gif");
ImageIcon icon = new ImageIcon(url);
```

Эти выражения означают: "искать файл `about.gif` там же, где находился файл `AboutPanel.class`".

Чтобы прочитать содержимое файла `about.txt`, можно использовать аналогичные команды:

```
URL url = AboutPanel.class.getResource("about.txt");
InputStream in = url.openStream();
```

Поскольку эта комбинация операторов очень широко распространена, удобно применить метод `getResourceAsStream()`, который возвращает объект класса `InputStream`, а не URL.

```
InputStream stream = AboutPanel.class.getResourceAsStream("about.txt");
```

Для того чтобы извлечь информацию из этого потока, нужно знать, как обрабатывается ввод (см. главу 12). В программе данные из потока читаются строка за строкой.

```
InputStream stream = AboutPanel.class.getResourceAsStream("about.txt");
Scanner in = Scanner.create(stream);
while (in.hasNext())
   textArea.append(in.nextLine() + "\n");
```

Среди файлов с демонстрационными программами из нашей книги есть JAR-файл `ResourceTest.jar`, содержащий все файлы классов для данного примера и файлы ресурсов `about.gif` и `about.txt`. Программа находит файл ресурсов вместе с файлами классов в одном и том же JAR-файле. Исходный текст этой программы содержится в листинге 10.12.

В предыдущем разделе говорилось, что изображения и аудиоданные можно извлекать с помощью методов `getImage()` и `getAudioClip()`. Эти же методы автоматически выполняют поиск в JAR-файлах. Однако, чтобы загрузить другие файлы из JAR-файла, нужно использовать метод `getResourceAsStream()`.

Ресурсы не обязательно помещать в один каталог с файлом классов. Вместо этого их можно записать в подкаталог. Имя ресурса с учетом подкаталогов выглядит так:

```
data/text/about.txt
```

Это — относительное имя ресурса, поэтому оно определяется относительно пакета, которому принадлежит класс, загружающий этот ресурс. Обратите внимание на то, что в этом имени всегда нужно использовать разделитель /, независимо от того, какие разделители применяются в операционной системе. Например, на платформе Windows загрузчик ресурсов автоматически преобразует символ / в разделитель \.

Имя ресурса, начинающееся с символа /, называется *абсолютным*. По этому имени ресурс обнаруживается точно так же, как и класс. Например, приведенный ниже ресурс находится в каталоге corejava.

```
/corejava/title.txt
```

Этот каталог может быть подкаталогом одного из каталогов, указанных в переменной окружения CLASSPATH, находиться в JAR-файле или, в случае аплета, на Web-сервере.

Несмотря на то что ресурс загружается автоматически, стандартных методов интерпретации содержимого файла ресурсов не существует. Каждый аплет по-своему интерпретирует данные, находящиеся в подобных файлах.

Кроме того, ресурсы применяются для интернационализации аплетов и приложений. В файлах ресурсов, например, содержатся сообщения и метки на разных языках. Каждый такой файл соответствует отдельному языку. Средства интернационализации, которые будут обсуждаются во втором томе в главе 10, поддерживают стандартный способ организации и доступа к этим файлам.

Листинг 10.12. Содержимое файла ResourceTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.swing.*;

public class ResourceTest
{
   public static void main(String[] args)
   {
      ResourceTestFrame frame = new ResourceTestFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
    Фрейм, содержащий панель с компонентами, которые
    демонстрируют доступ к ресурсам из JAR-файла.

*/
class ResourceTestFrame extends JFrame
{
   public ResourceTestFrame()
   {
      setTitle("ResourceTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      add(new AboutPanel());
   }
```

```
   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 300;
}

/**
      Панель с текстовой областью и кнопкой About. После щелчка
      на кнопке текстовая область заполняется текстом,
      полученным из файла ресурса.
*/
class AboutPanel extends JPanel
{
   public AboutPanel()
   {
      setLayout(new BorderLayout());

      // Добавление текстовой области.
      textArea = new JTextArea();
      add(new JScrollPane(textArea), BorderLayout.CENTER);

      // Добавление кнопки About
      URL aboutURL = AboutPanel.class.getResource("about.gif");
      JButton aboutButton = new JButton("About",
         new ImageIcon(aboutURL));
      aboutButton.addActionListener(new AboutAction());
      add(aboutButton, BorderLayout.SOUTH);
   }

   private JTextArea textArea;

   private class AboutAction implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         InputStream stream =
            AboutPanel.class.getResourceAsStream("about.txt");
         Scanner in = new Scanner(stream);
         while (in.hasNext())
            textArea.append(in.nextLine() + "\n");
      }
   }
}
```

java.lang.Class 1.0

- `URL getResource(String name)`
- `InputStream getResourceAsStream(String name)` 1.1

Находит ресурс там же, где расположен класс, и возвращает его URL или поток ввода, который можно использовать для загрузки. Если ресурс не найден, методы возвращают ссылку `null`, не генерируя исключений и не сообщая об ошибках ввода-вывода.

Параметр name Имя ресурса

Герметизация пакета

В главе 4 уже упоминалось, что Java-пакет можно "герметизировать", чтобы в него больше нельзя было добавить новые классы. Разработчики обычно применяют герметизацию в том случае, если они используют классы, методы и поля, имеющие область видимости, ограниченную пакетом.

Например, если вы герметизируете пакет `com.mycompany.util`, то приведенное ниже выражение нельзя будет применить ни для одного класса вне герметизированного архива.

```
package com.mycompany.util;
```

По умолчанию пакеты в JAR-файле не герметизированы. Изменить данные установки можно, поместив в главный раздел файла манифеста следующую строку:

```
Sealed: true
```

Для каждого пакета можно отдельно указать, герметизировать его или нет, добавив в файл манифеста новый раздел.

```
Name: com/mycompany/util/
Sealed: true
Name: com/mycompany/misc/
Sealed: false
```

Для того чтобы герметизировать пакет, нужно создать текстовый файл с инструкциями манифеста, а затем выполнить команду

```
jar cvmf MyArchive.jar manifest.mf добавляемые файлы
```

Технология Java Web Start

Java Web Start – это новая технология, предназначенная для того, чтобы упростить работу с программами, доставляемыми по глобальной сети. Между технологией Java Web Start и аплетами существуют принципиальные различия.

- Технология Java Web Start используется для доставки обычных приложений, написанных на языке Java и начинающихся с вызова метода `main()`, который содержится в одном из классов.
- Приложения, созданные с помощью технологии Java Web Start, не запускаются в среде браузера. Их интерфейсные элементы отображаются вне браузера.
- Приложения, созданные с помощью технологии Java Web Start, можно получить с помощью браузера, однако механизм, лежащий в основе этого процесса, совершенно отличен от получения аплетов. Браузеры тесно связаны с системой поддержки выполнения программ, написанных на языке Java, которая запускает аплеты. Технология Java Web Start предоставляет гораздо большую свободу. Браузер запускает внешнюю программу при получении дескриптора приложения. Для этого используется механизм, подобный тому, который применяется в Adobe Acrobat или RealAudio.
- Скопированное приложение Java Web Start запускается независимо от браузера.
- На приложения Java Web Start накладываются гораздо меньшие ограничения, чем на аплеты. В частности, приложения, лишенные цифровой подписи, могут получать доступ к некоторым локальным ресурсам.

Для того чтобы подготовить приложение к доставке с помощью технологии Java Web Start, необходимо запаковать его в один или несколько JAR-файлов. Затем нужно подготовить файл описания в формате JNLP (Java Network Launch Protocol – сетевой протокол запуска Java-приложений). Теперь разместите файлы на Web-сервере. После этого необходимо убедиться, что на Web-сервере MIME-тип `application/x-java-jnlp-file` связан с расширением `.jnlp`. (Браузеры используют MIME-тип для того, чтобы определить, какую программу следует запустить для поддержки документа.) Детали можно найти в документации, сопровождающей ваш Web-сервер.

Чтобы поэкспериментировать с технологией Java Web Start, инсталлируйте программу Tomcat. Найти ее можно по адресу `jakarta.apache.org/tomcat`. Tomcat — это контейнер для сервлетов и JSP-документов, но он также поддерживает и Web-страницы. Текущая версия этого контейнера настроена на обработку JNLP-файлов.

Попробуйте доставить с помощью технологии Java Web Start приложение, реализующее калькулятор, которое было описано в главе 9. Для этого выполните следующие действия.

1. Скомпилируйте программу `Calculator.java`.
2. Подготовьте файл манифеста `Calculator.mf`, содержащий следующую строку.

   ```
   Main-Class: Calculator
   ```

3. Создайте JAR-файл с помощью команды

   ```
   jar cvfm Calculator.jar Calculator.mf *.class
   ```

4. Подготовьте файл `Calculator.jnlp`, содержимое которого приведено ниже.

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <jnlp spec="1.0+"codebase="http://localhost:8080/calculator/"
        href="Calculator.jnlp">
     <information>
       <title>Calculator Demo Application</title>
       <vendor>Cay S. Horstmann</vendor>
       <description>A Calculator</description>
       <offline-allowed/>
     </information>
     <resources>
       <j2se version="5.0+"/>
       <jar href="Calculator.jar"/>
     </resources>
     <application-desc/>
   </jnlp>
   ```

   Формат файла очевиден. Его спецификация содержится на Web-странице `http://java.sun.com/products/javawebstart/docs/developersguide.html`.

5. Если вы используете контейнер Tomcat, создайте каталог *tomcat*`/webapps/calculator`, где *tomcat* – каталог, в котором инсталлирован продукт Tomcat. Создайте подкаталог `tomcat/webapps/calculator/WEB-INF` и поместите в него минимальную версию файла `web.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
</web-app>
```

6. Разместите JAR- и JNLP-файл на Web-сервере так, чтобы URL соответствовал записи codebase в JNLP-файле. Если вы используете сервер Tomcat, это будет каталог *tomcat*/webapps/calculator.
7. Убедитесь, что браузер правильно настроен для использования средств Java Web Start, проверив, что MIME-тип application/x-java-jnlp-file связан с приложением javaw. Если на вашем компьютере инсталлирован пакет JDK, настройка выполняется автоматически.
8. Запустите Tomcat.
9. Обратитесь с помощью браузера к файлу http://localhost:8080/calculator/Calculator.jnlp.
10. Вы должны увидеть окно, подобное тому, которое показано на рис. 10.12. Затем на экране должен появиться калькулятор, обрамление которого свидетельствует о том, что он является приложением Java Web Start (рис. 10.13).
11. Когда вы снова обратитесь к файлу JNLP, приложение будет извлечено из кэша. Просмотреть содержимое кэша можно с помощью панели управления Java Plug-in (рис. 10.14). Начиная с версии JDK 5.0, панель управления используется как для аплетов, так и для приложенрий Java Web Start. Если вы работаете в системе Windows, панель управления Java Plug-in можно найти на системной панели управления. В системе Linux следует запустить jdk/jre/bin/ControlPanel.

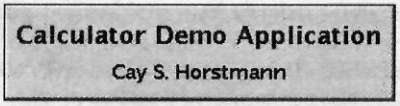


Рис. 10.12. Доставка приложения средствами Java Web Start

Рис. 10.13. Приложение Calculator, доставленное с помощью Java Web Start

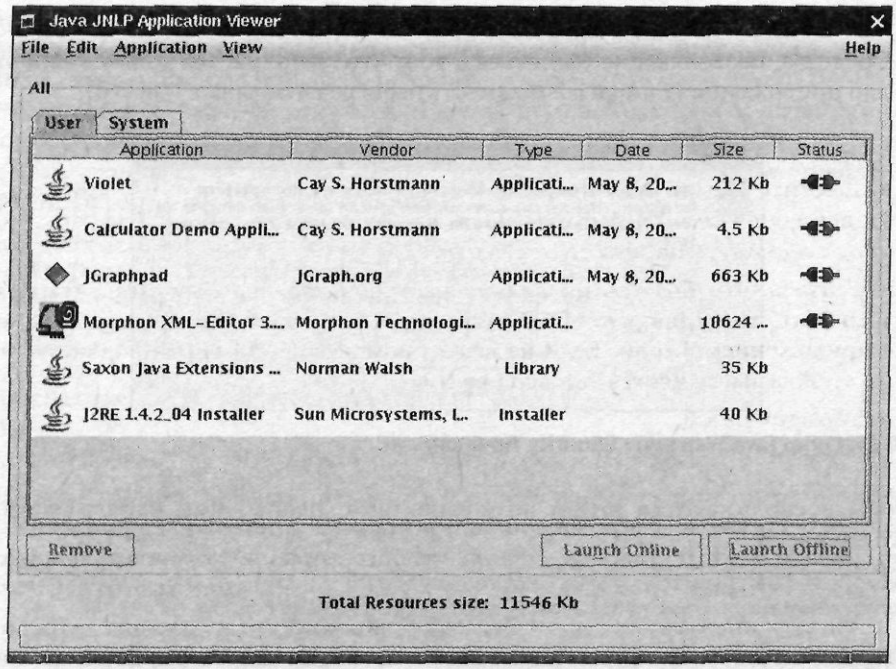


Рис. 10.14. Просмотр кэша

JNLP API

Для того чтобы приложение Java Web Start получило полный доступ к локальному компьютеру, оно должно иметь электронную подпись. (Электронные подписи рассматриваются в главе 9 второго тома.) Неподписанный аплет, загруженный из сети, представляет опасность для локального компьютера и запускается с жесткими ограничениями. Однако, предоставляя минимальные привилегии, JNLP API все же предоставляет инструменты для доступа к некоторым локальным ресурсам.

Например, приложение имеет ограниченные возможности чтения и записи файлов. Программа не может просматривать файловую систему и указывать имена файлов. Вместо этого на экране отобразится диалоговое окно, в котором пользователь может выбрать требуемый файл. Перед появлением диалогового окна для выбора файлов пользователь получает предупреждение и должен выразить согласие на продолжение работы (рис. 10.15). Более того, на самом деле API не предоставляет программе доступа к объекту `File`. В частности, приложение не может самостоятельно обнаружить файл. Системная информация максимально скрыта от сомнительных приложений.

API предоставляет разработчику следующие возможности.

- Чтение и запись файлов.
- Доступ к клавиатуре.
- Копирование файла.
- Печать.
- Хранение и извлечение конфигурационных данных.

- Отображение документа в браузере, заданном по умолчанию.
- Средства, позволяющие убедиться в том, что выполняется только один экземпляр приложения (данная возможность была реализована в JDK 5.0).

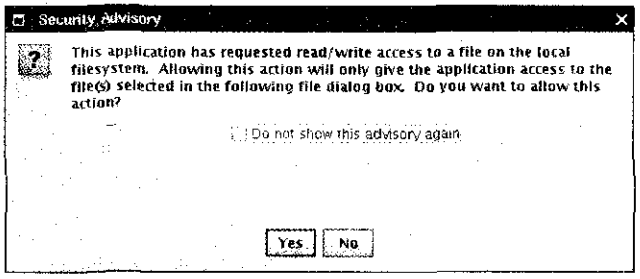


Рис. 10.15. Окно Java Web Start Security Advisory

Чтобы воспользоваться этими возможностями, необходимо использовать класс `ServiceManager`.

```
FileSaveService service = (FileSaveService)
   ServiceManager.lookup("javax.jnlp.FileSaveService");
```

Если соответствующая функция оказалась недоступной, генерируется исключение `UnavailableServiceException`.

Для компиляции программы, использующей интерфейс JNLP API, необходимо указать в составе пути для поиска классов файл `javaws.jar`. Этот файл находится в подкаталоге `jre/lib` каталога JDK

Теперь перейдем к обсуждению наиболее полезных возможностей JNLP. Чтобы сохранить файл, следует указать в диалоговом окне предположения о первоначальном пути и расширения файлов, а также сохраняемые данные и предполагаемое имя файла. Рассмотрим пример:

```
service.saveFileDialog(".", new String[] { "txt" },
   data, "calc.txt");
```

Данные должны быть доставлены в поток `InputStream`. Иногда эта задача оказывается довольно сложной. Программа, приведенная в листинге 10.13, использует описанную ниже стратегию.

1. Создается экземпляр класса `ByteArrayOutputStream`, предназначенный для хранения байтов, подлежащих записи на диск.
2. Создается экземпляр класса `PrintStream`, посылающий эти данные в поток `ByteArrayOutputStream`.
3. Информация, подлежащая сохранению в потоке печати, выводится в `PrintStream`.
4. Создается экземпляр класса `ByteArrayInputStream`, считывающий сохраненные байты.
5. Поток передается методу `saveFileDialog()`.

Более подробно потоки описываются в главе 12. Пока же мы можем не обращать внимание на детали, указанные в демонстрационной программе.

Чтобы считать данные из файла, используется класс `FileOpenService`. Его метод `openFileDialog()` получает первоначальный путь и расширения файлов и возвращает объект класса `FileContents`. Затем можно вызвать метод `getInputStream()` и считать данные из файла. Если пользователь не выбрал файл, метод `openFileDialog()` возвращает значение `null`.

```
FileOpenService service = (FileOpenService)
   ServiceManager.lookup("javax.jnlp.FileOpenService");
FileContents contents =  service.openFileDialog(".",
   new String[] { "txt" });
if (contents != null)
{
   InputStream in = contents.getInputStream();
   . . .
}
```

Для того чтобы отобразить документ в браузере по умолчанию (аналогично методу `showDocument()` для аплетов), применяется интерфейс `BasicService`. Учтите, что в некоторых системах (в частности, во многих версиях Unix и Linux) браузер по умолчанию отсутствует.

```
BasicService service = (BasicService)
   ServiceManager.lookup("javax.jnlp.BasicService");
if (service.isWebBrowserSupported())
   service.showDocument(url);
else ...
```

Существует очень устаревший метод `PersistentService()`, позволяющий приложению сохранять небольшие объемы информации о настройках и извлекать ее, если приложение запускается вновь. Эта возможность необходима, поскольку приложение, источник которого вызывает сомнение, не может задавать местоположение конфигурационного файла.

Средства сохранения информации о настройках напоминают механизм cookie, применяемый при работе с протоколом HTTP. В качестве ключей используются URL. Эти URL не обязательно должны ссылаться на реальные ресурсы. Они лишь представляют собой удобную схему именования файлов. По каждому URL-ключу приложение может записать произвольные двоичные данные. (Объем записи может быть ограничен размером блока.)

Для того чтобы изолировать приложения друг от друга, каждое приложение должно использовать URL, начинающиеся с имени сервера, содержащего его код (указанный в JNLP-файле). Например, если приложение загружено с Web-страницы `http://myserver.com/apps`, то оно может использовать лишь ключи, имеющие вид `http://myserver.com/apps/subkey1/subkey2/...`. Попытка получить доступ к другим ключам будет заведомо безуспешной.

Чтобы определить адрес сервера, на котором хранится код, приложение может вызвать метод `getCodebase()` класса `BasicService`.

Новый ключ создается с помощью метода `create()` класса `PersistenceService`.

```
URL url = new URL(codebase, "mykey");
servise.create(url, maxSize);
```

Для доступа к информации, связанной с конкретным ключом, вызывается метод `get()`. Этот метод возвращает экземпляр класса `FileContents`, с помощью которого можно читать и записывать данные, соответствующие указанному ключу.

```
FileContents contents = service.get(url);
InputStream in = contents.getInputStream();
OutputStream out = contents.getOutputStream(true);
   // true = перезапись
```

К сожалению, определить, существует указанный ключ или его нужно создать заново, довольно сложно. Мы можем лишь надеяться, что ключ существует, и вызывать метод `get`. Если при этом генерируется исключение `FileNotFoundException`, необходимо создать новый ключ.

Начиная с JDK 5.0, и приложения Java Web Start, и аплеты могут выводить данные на печать, используя обычные средства API. При этом отображается диалоговое окно, запрашивающее у пользователя согласие на доступ к принтеру. Подробно средства API для работы с принтером будут рассматриваться во втором томе.

Программа, код которой приведен в листинге 10.13, *представляет собой модификацию* приложения, реализующего калькулятор. Этот калькулятор имеет виртуальную бумажную ленту, на которую записываются результаты всех вычислений. Список всех предыдущих вычислений можно сохранять и считывать из файла. Для демонстрации постоянного хранения данных приложение позволяет задавать заголовок фрейма. Если программу запустить вновь, она извлечет этот заголовок из постоянного хранилища (рис. 10.16).

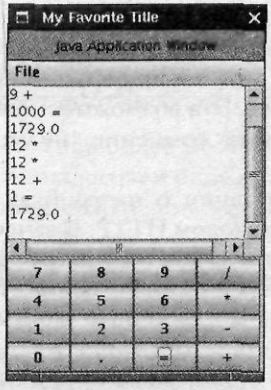


Рис. 10.16. Приложение `WebStartCalculator`

Листинг 10.13. Содержимое файла `WebStartCalculator.java`

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.*;
import javax.swing.text.*;
import javax.jnlp.*;
```

```
/**
   Калькулятор с поддержкой предыстории. Может быть
   доставлен средствами Java Web Start.
*/
public class WebStartCalculator
{
   public static void main(String[] args)
   {
      CalculatorFrame frame = new CalculatorFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
    Фрейм, содержащий панель калькулятора и меню
    для чтения и записи предыдущих результатов.
*/
class CalculatorFrame extends JFrame
{
   public CalculatorFrame()
   {
      setTitle();
      panel = new CalculatorPanel();
      add(panel);

      JMenu fileMenu = new JMenu("File");

      JMenuItem openItem = fileMenu.add("Open");
      openItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               open();
            }
         });

      JMenuItem saveItem = fileMenu.add("Save");
      saveItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               save();
            }
         });
      JMenuBar menuBar = new JMenuBar();
      menuBar.add(fileMenu);
      setJMenuBar(menuBar);

      pack();
   }

   /**
       Извлечение заголовка из постоянного хранилища.
```

```
      Если ранее заголовок не был задан, он запрашивается
      у пользователя.
   */
   public void setTitle()
   {
      try
      {
         String title = null;

         BasicService basic = (BasicService)
            ServiceManager.lookup("javax.jnlp.BasicService");
         URL codeBase = basic.getCodeBase();

         PersistenceService service = (PersistenceService)
            ServiceManager.lookup
            ("javax.jnlp.PersistenceService");
         URL key = new URL(codeBase, "title");

         try
         {
            FileContents contents = service.get(key);
            InputStream in = contents.getInputStream();
            BufferedReader reader = new BufferedReader(new
               InputStreamReader(in));
            title = reader.readLine();
         }
         catch (FileNotFoundException e)
         {
            title = JOptionPane.showInputDialog
               ("Please supply a frame title:");
            if (title == null) return;

            service.create(key, 100);
            FileContents contents = service.get(key);
            OutputStream out = contents.getOutputStream(true);
            PrintStream printOut = new PrintStream(out);
            printOut.print(title);
         }
         setTitle(title);
      }
      catch (UnavailableServiceException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
      catch (MalformedURLException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   /**
```

```
   Открытие файла с предысторией и обновление отображаемых
   данных.
*/
public void open()
{
   try
   {
      FileOpenService service = (FileOpenService)
         ServiceManager.lookup("javax.jnlp.FileOpenService");
      FileContents contents = service.openFileDialog(".",
         new String[] { "txt" });

      JOptionPane.showMessageDialog(this, contents.getName());
      if (contents != null)
      {
         InputStream in = contents.getInputStream();
         BufferedReader reader = new BufferedReader(new
            InputStreamReader(in));
         String line;
         while ((line = reader.readLine()) != null)
         {
            panel.append(line);
            panel.append("\n");
         }
      }
   }
   catch (UnavailableServiceException e)
   {
      JOptionPane.showMessageDialog(this, e);
   }
   catch (IOException e)
   {
      JOptionPane.showMessageDialog(this, e);
   }
}

/**
   Сохранение предыстории в файле.
*/
public void save()
{
   try
   {
      ByteArrayOutputStream out = new ByteArrayOutputStream();
      PrintStream printOut = new PrintStream(out);
      printOut.print(panel.getText());
      InputStream data = new
         ByteArrayInputStream(out.toByteArray());
      FileSaveService service = (FileSaveService)
         ServiceManager.lookup("javax.jnlp.FileSaveService");
      service.saveFileDialog(".",
         new String[] { "txt" }, data, "calc.txt");
   }
   catch (UnavailableServiceException e)
   {
      JOptionPane.showMessageDialog(this, e);
```

```
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   private CalculatorPanel panel;
}


/**
   Панель с кнопками калькулятора и областью для отображения
   результатов.
*/
class CalculatorPanel extends JPanel
{
   /**
      Компоновка панели.
   */
   public CalculatorPanel()
   {
      setLayout(new BorderLayout());

      result = 0;
      lastCommand = "=";
      start = true;

      // Добавление области для отображения данных.

      display = new JTextArea(10, 20);

      add(new JScrollPane(display), BorderLayout.NORTH);

      ActionListener insert = new InsertAction();
      ActionListener command = new CommandAction();

      // Размещение кнопок в виде сетки 4x4

      panel = new JPanel();
      panel.setLayout(new GridLayout(4, 4));

      addButton("7", insert);
      addButton("8", insert);
      addButton("9", insert);
      addButton("/", command);

      addButton("4", insert);
      addButton("5", insert);
      addButton("6", insert);
      addButton("*", command);

      addButton("1", insert);
      addButton("2", insert);
      addButton("3", insert);
      addButton("-", command);
```

```
      addButton("0", insert);
      addButton(".", insert);
      addButton("=", command);
      addButton("+", command);

      add(panel, BorderLayout.CENTER);
   }

   /**
      Получение текста предыстории.
      @return Предыстория работы калькулятора
   */
   public String getText()
   {
      return display.getText();
   }

   /**
      Добавление строки к тексту предыстории.
      @param s Добавляемая строка
   */
   public void append(String s)
   {
      display.append(s);
   }

   /**
      Добавление кнопки на центральную панель.
      @param label Метка кнопки
      @param listener Обработчик событий, связанных с кнопкой
   */
   private void addButton(String label, ActionListener listener)
   {
      JButton button = new JButton(label);
      button.addActionListener(listener);
      panel.add(button);
   }

   /**
      Данный обработчик добавляет строку в конец отображаемого
      текста.
   */
   private class InsertAction implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         String input = event.getActionCommand();
         start = false;
         display.append(input);
      }
   }

   /**
      Данный обработчик выполняет команду, соответствующую
      кнопке.
```

```
*/
private class CommandAction implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      String command = event.getActionCommand();

      if (start)
      {
         if (command.equals("-"))
         {
            display.append(command);
            start = false;
         }
         else
            lastCommand = command;
      }
      else
      {
         try
         {
            int lines = display.getLineCount();
            int lineStart =
               display.getLineStartOffset(lines - 1);
            int lineEnd = display.getLineEndOffset(lines - 1);
            String value = display.getText(lineStart,
               lineEnd - lineStart);
            display.append(" ");
            display.append(command);
            calculate(Double.parseDouble(value));
            if (command.equals("="))
               display.append("\n" + result);
            lastCommand = command;
            display.append("\n");
            start = true;
         }
         catch (BadLocationException e)
         {
            e.printStackTrace();
         }
      }
   }
}

/**
   Выполняет вычисления.
   @param x Значение, обрабатываемое с предыдущим результатом
*/
public void calculate(double x)
{
   if (lastCommand.equals("+")) result += x;
   else if (lastCommand.equals("-")) result -= x;
   else if (lastCommand.equals("*")) result *= x;
   else if (lastCommand.equals("/")) result /= x;
   else if (lastCommand.equals("=")) result = x;
}
```

```
   private JTextArea display;
   private JPanel panel;
   private double result;
   private String lastCommand;
   private boolean start;
}
```

javax.jnlp.ServiceManager

- `static String[] getServiceNames()`

  Возвращает имена объектов, обеспечивающих все доступные функциональные возможности.

- `static Object lookup(String name)`

  Возвращает объект, обеспечивающий одну из функциональных возможностей и имеющий заданное имя.

javax.jnlp.BasicService

- `URL getCodeBase()`

  Возвращает имя каталога, содержащего код данного приложения.

- `boolean isWebBrowserSupported()`

  Возвращает значение `true`, если среда Java Web Start может запустить Web-браузер.

- `boolean showDocument(URL url)`

  Пытается отобразить заданный ресурс с помощью браузера. Если запрос выполнен успешно, возвращает значение `true`.

javax.jnlp.FileContents

- `inputStream getInputStream()`

  Возвращает входной поток, предназначенный для чтения содержимого файла.

- `OutputStream getOutputStream(boolean overwrite)`

  Возвращает выходной поток, предназначенный для записи в файл; если значение параметра `overwrite` равно `true`, содержимое файла перезаписывается.

- `String getName()`

  Возвращает имя файла (но не полный путь).

- `boolean canRead()`
- `boolean canWrite()`

  Эти методы возвращают значение `true`, если соответствующий файл можно прочитать или записать.

javax.jnlp.FileOpenService

- `FileContents openFileDialog(String pathHint, String[] extensions)`

- `FileContents openMultiDialog(String pathHint, String[] extensions)`

Эти методы выводят предупреждающее сообщение и диалоговое окно для выбора файлов. Они возвращают дескрипторы файлов, выбранных пользователем, или значение `null`, если пользователь не выбрал ни одного файла.

javax.jnlp.FileSaveService

- `FileContents saveFileDialog(String pathHint, String[] extensions, String nameHint)`
- `FileContents saveFileDialog(String pathHint, String[] extensions, FileContents data)`

Эти методы выводят предупреждающее сообщение и диалоговое окно для выбора файлов. Они записывают данные и возвращают дескрипторы файлов, выбранных пользователем, или значение `null`, если пользователь не выбрал ни одного файла.

javax.jnlp.PersistenceService

- `long create(URL key, long maxsize)`

Создает в постоянном хранилище запись, идентифицируемую заданным ключом. Возвращает максимальный размер памяти, предоставленной элементу.

- `void delete(URL key)`

Удаляет запись, соответствующую заданному ключу.

- `String[] getNames(URL url)`

Возвращает относительные имена всех ключей, начинающихся с заданного URL.

- `FileContents get(URL key)`

Определяет дескриптор, с помощью которого можно модифицировать данные, связанные с заданным ключом. Если заданному ключу ничего не соответствует, генерируется исключение `FileNotFoundException`.

Хранение параметров настройки

Большинство программ могут настраиваться пользователями. Затем эти программы должны сохранить выполненные установки, чтобы учесть их при следующем запуске. Мы уже видели, что приложения, созданные по технологии Java Web Start, могут использовать постоянные хранилища. Однако в JDK 1.4 был реализован иной, более мощный механизм для локальных приложений. Мы опишем этот механизм, но сначала рассмотрим более простой, традиционный подход, позволяющий хранить информацию о настройках приложений, написанных на языке Java.

Карты свойств

Карта свойств (property map) – это структура данных, в которой хранятся пары ключ–значение. Карты свойств часто используются для хранения информации о параметрах настройки и характеризуются тремя особенностями.

- Ключи и значения являются строками.
- Всю структуру легко записать в файл и прочитать оттуда.
- Существует вспомогательная таблица для значений, принимаемых по умолчанию.

На платформе Java класс, реализующий карту свойств, называется `Properties`. Карты свойств используются при настройке программ. Рассмотрим пример.

```
Properties settings = new Properties();
settings.put("font", "Courier");
settings.put("size", "10");
settings.put("message", "Hello, World");
```

Для записи списка свойств в файл используется метод `store()`. В данном случае мы просто выведем множество свойств в стандартный поток вывода. Второй параметр представляет собой комментарии, которые помещаются в файл.

```
FileOutputStream  out
   = new FileOutputStream("Myprog.properties");
settings.store(out, "Myprog Properties");
```

В результате выполнения приведенного фрагмента будут выведены следующие данные:

```
#Myprog Properties
#Sun Jul 14 07:22:52  2002
font=Courier
size=10
message=Hello,World
```

Чтобы загрузить эти свойства из файла, следует выполнить следующие выражения:

```
FileInputStream in
   = new FileInputStream("Myprog.properties");
settings.load(in);
```

Применим этот подход, чтобы дать пользователю возможность настраивать программу `NotHelloWorld` по своему усмотрению. Для этого позволим ему задавать следующие параметры в конфигурационном файле `CustomWorld.properties`.

- Размер окна.
- Название шрифта.
- Размер шрифта.
- Цвет фона.
- Строка сообщения.

Если пользователь не указал какой-либо параметр, он будет задан по умолчанию.

Класс `Properties` имеет два механизма для указания параметров по умолчанию. Во-первых, всякий раз, извлекая строку, можно задать значение по умолчанию. Оно будет использоваться автоматически, если соответствующий ключ не указан.

```
String font = setings.getProperty("font", "Courier");
```

Если в таблице есть свойство `"font"`, имя шрифта задается указанной строкой. В противном случае в качестве шрифта по умолчанию будет использоваться шрифт `"Courier"`.

Если пользователь сочтет, что задавать значения по умолчанию при каждом вызове метода `getProperty()` слишком утомительно, он может упаковать все значения, принятые по умолчанию, во вспомогательную таблицу свойств и передать ее конструктору искомой таблицы.

```
Properties defaultSettings = new Properties();
defaultSettings.put("font", "Courier");
defaultSettings.put("size", "10");
defaultSettings.put("color.red", "0");
...
Properties settings = new Properties(defaultSettings);
FileStream in
   = new FileInputStream("CustomWorld.properties");
settings.load(in);
...
```

В листинге 10.14 показан код настраиваемой программы `"Hello, Custom World"`. Для того чтобы изменить внешний вид этой программы (рис. 10.17) по своему усмотрению, достаточно отредактировать файл с расширением `.properties`.

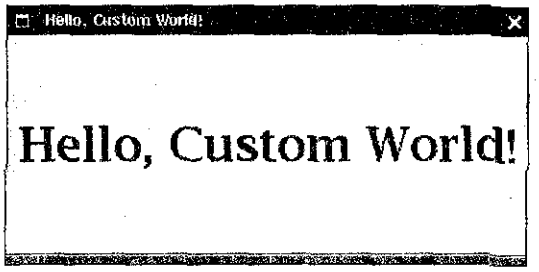


Рис. 10.17. Настраиваемая программа Hello, Custom World!

Свойства хранятся в простой таблице, не имеющей иерархической структуры. Как правило, используется фиктивная иерархия с именами ключей `color.red`, `color.green` и т.п. Однако в классе `Properties` нет методов, позволяющих создать настоящую иерархию. Если в файле хранится сложная информация о конфигурации программы, следует использовать класс `Preferences`, который будет описан в следующем разделе.

Класс `Properties` расширяет класс `Hashtable`. Это значит, что все методы класса `Hashtable` доступны объектам класса `Properties`. Некоторые из них очень полезны. Например, метод `size()` возвращает количество возможных свойств (но не всех — свойства, заданные по умолчанию, он не учитывает). Аналогично, метод `keys()` возвращает объект `Enumeration`, содержащий все ключи, за исключением тех, которые принимаются по умолчанию. Существует еще один метод, `propertyNames()`, возвращающий все ключи. Метод `put()` представляет очевидную опасность. Он не проверяет, какие строки вы помещаете в таблицу.

Существует ли между этими классами отношение "является" — основание для наследования? Каждая ли карта свойств является таблицей хэширования? Конечно, нет.

Об этом со всей очевидностью свидетельствуют детали реализации. Возможно, было бы лучше считать множество свойств классом, содержащим таблицу хэширования. Но в этом случае таблица хэширования должна была бы быть закрытой переменной. На самом деле множество свойств использует две таблицы хэширования: одну — для значений, задаваемых по умолчанию, а вторую — для других значений.

Мы считаем, что класс `Properties` следовало бы организовать иначе.

```
class Properties
{
   public String getProperty(String) { ... }
   public void put(String, String) { ... }
   ...
   private Hashtable nonDefaults;
   private Hashtable defaults;
}
```

Мы вовсе не призываем игнорировать класс `Propertiers`. Если проявлять осторожность и не вводить ничего, кроме строк, он работает просто прекрасно. Но следует дважды подумать, прежде чем использовать этот неудачно примененный механизм наследования в своей программе.

Листинг 10.14. Содержимое файла `CustomWorld.java`

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import javax.swing.*;

/**
    Эта программа демонстрирует настройку
    с помощью файла свойств.
*/
public class CustomWorld
{
   public static void main(String[] args)
   {
      CustomWorldFrame frame = new CustomWorldFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
      В этом фрейме выводится сообщение. Размер фрейма,
      текст сообщения, шрифт и цвет задаются в файле свойств.
*/
class CustomWorldFrame extends JFrame
{
   public CustomWorldFrame()
   {
      Properties defaultSettings = new Properties();
      defaultSettings.put("font", "Monospaced");
      defaultSettings.put("width", "300");
      defaultSettings.put("height", "200");
      defaultSettings.put("message", "Hello, World");
      defaultSettings.put("color.red", "0 50 50");
```

```
      defaultSettings.put("color.green", "50");
      defaultSettings.put("color.blue", "50");
      defaultSettings.put("ptsize", "12");

      Properties settings = new Properties(defaultSettings);
      try
      {
         FileInputStream in = new
            FileInputStream("CustomWorld.properties");
         settings.load(in);
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }

      int red = Integer.parseInt
         (settings.getProperty("color.red"));
      int green = Integer.parseInt
         (settings.getProperty("color.green"));
      int blue = Integer.parseInt
         (settings.getProperty("color.blue"));

      Color foreground = new Color(red, green, blue);

      String name = settings.getProperty("font");
      int ptsize = Integer.parseInt
         (settings.getProperty("ptsize"));
      Font f = new Font(name, Font.BOLD, ptsize);

      int hsize =
         Integer.parseInt(settings.getProperty("width"));
      int vsize =
         Integer.parseInt(settings.getProperty("height"));
      setSize(hsize, vsize);
      setTitle(settings.getProperty("message"));

      JLabel label = new
         JLabel(settings.getProperty("message"),
         SwingConstants.CENTER);
      label.setFont(f);
      label.setForeground(foreground);
      add(label);
   }
}
```

java.util.Properties 1.0

- `Properties()`

  Создает пустой список свойств.

- `Properties(Properties defaults)`

  Создает пустой список свойств со значениями, заданными по умолчанию.

  Параметр `defaults` Значения, заданные по умолчанию

- `String getProperty(String key)`

  Возвращает строку, связанную с заданным ключом, или, если ключ в основной таблице отсутствует, строку, связанную с ключом из таблицы значений, принятых по умолчанию. Если указанного ключа в таблице значений, принятых по умолчанию, тоже нет, возвращается значение `null`.

Параметр	`key`	Ключ, связанный с возвращаемой строкой

- `String getProperty(String key, String defaultValue)`

  Возвращает строку, связанную с указанным ключом. Если ключа в таблице нет, возвращает свойство, заданное по умолчанию.

Параметры	`key`	Ключ, связанный с возвращаемой строкой
	`defaultValue`	Строка, которая возвращается, если ключа в таблице нет

- `void load(InputStream in) throws IOException`

  Загружает карту свойств из входного потока.

Параметр	`in`	Входной поток

- `void store(OutputStream out, String header)` **1.2**

  Записывает карту свойств в выходной поток.

Параметры	`out`	Выходной поток
	`header`	Заголовок, помещаемый в первую строку файла

Информация о системе

Рассмотрим еще один пример, демонстрирующий универсальность класса `Properties`. Информация об операционной системе хранится в объекте `Properties`, который возвращается методом `getProperties()` класса `System`.

```
Properties sysprops = System.getProperties();
```

Чтобы получить доступ к определенному свойству, нужно вызвать следующий метод:

```
String value = System.getProperty(key);
```

Приложения, которые запускаются без участия диспетчера защиты, имеют полный доступ к этой информации, однако аплетам и другие сомнительным программам доступны только перечисленные ниже ключи.

```
java.version
java.vendor
java.vendor.url
java.class.version
os.name
os.version
os.arch
file.separator
path.separator
line.separator
java.specification.version
```

```
java.vm.specification.version
java.vm.specification.vendor
java.vm.specification.name
java.vm.version
java.vm.vendor
java.vm.name
```

Если аплет пытается прочитать информацию, соответствующую другому ключу (или все свойства), генерируется исключение.

Имена доступных свойств системы указаны в файле `security/java.policy`, который расположен в том же каталоге, что и средства поддержки выполнения программ.

Программа, приведенная в листинге 10.15, выводит пары ключ–свойство, хранящиеся в объекте `Properties`, содержащем информацию о системе.

Ниже приведены строки, полученные в результате работы этой программы. (Разумеется, содержание этих строк зависит от конкретных настроек компьютера.)

```
#System Properties
#Sat May 08 08:27:34 PDT 2004
java.runtime.name=Java(TM) 2 Runtime Environment, Standard Edition
sun.boot.library.path=/usr/local/jdk5.0/jre/lib/i386
java.vm.version=5.0
java.vm.vendor=Sun Microsystems Inc.
java.vendor.url=http\://java.sun.com/
path.separator=\:
java.vm.name=Java HotSpot(TM) Client VM
file.encoding.pkg=sun.io
user.country=US
sun.os.patch.level=unknown
java.vm.specification.name=Java Virtual Machine Specification
user.dir=/home/cay/books/cj5/corejava/v1/v1ch10/SystemInfo
java.runtime.version=5.0
java.awt.graphicsenv=sun.awt.X11GraphicsEnvironment
java.endorsed.dirs=/usr/local/jdk5.0/jre/lib/endorsed
os.arch=i386
java.io.tmpdir=/tmp
line.separator=\n
java.vm.specification.vendor=Sun Microsystems Inc.
os.name=Linux
java.library.path=/usr/local/jdk5.0/jre/lib/i386/client\:/usr/local/jdk5.0/jre/
lib/i386\:/usr/local/jdk5.0/jre/../lib/i386
java.specification.name=Java Platform API Specification
java.class.version=48.0
sun.management.compiler=HotSpot Client Compiler
java.util.prefs.PreferencesFactory=java.util.prefs.FileSystemPreferencesFactory
os.version=2.4.20-8
user.home=/home/cay
user.timezone=America/Los_Angeles
java.awt.printerjob=sun.print.PSPrinterJob
file.encoding=UTF-8
java.specification.version=5.0
java.class.path=.
user.name=cay
java.vm.specification.version=1.0
```

```
java.home=/usr/local/jdk5.0/jre
sun.arch.data.model=32
user.language=en
java.specification.vendor=Sun Microsystems Inc.
java.vm.info=mixed mode
java.version=5.0
java.ext.dirs=/usr/local/jdk5.0/jre/lib/ext
sun.boot.class.path=/usr/local/jdk5.0/jre/lib/rt.jar\:/usr/local/jdk5.0/jre/lib
/i18n.jar\;/usr/local/jdk5.0/jre/lib/sunrsasign.jar\:/usr/local/jdk5.0/jre/lib/
jsse.jar\:/usr/local/jdk5.0/jre/lib/jce.jar\:/usr/local/jdk5.0/jre/lib/charsets
.jar\:/usr/local/jdk5.0/jre/classes
java.vendor=Sun Microsystems Inc.
file.separator=/
java.vendor.url.bug=http\://java.sun.com/cgi-bin/bugreport.cgi
sun.io.unicode.encoding=UnicodeLittle
sun.cpu.endian=little
sun.cpu.isalist=
```

Листинг 10.15. Содержимое файла `SystemInfo.java`

```
import java.applet.*;
import java.io.*;
import java.util.*;

/**
   Данная программа отображает системные свойства.
*/
public class SystemInfo
{
   public static void main(String args[])
   {
      try
      {
         Properties sysprops = System.getProperties();
         sysprops.store(System.out, "System Properties");
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }
}
```

`java.lang.System 1.0`

- `Properties getProperties()`

  Извлекает все системные свойства. Приложение должно иметь доступ ко всем свойствам, в противном случае генерируется исключение.

- `String getProperty(String key)`

  Извлекает системное свойство, связанное с заданным ключом. Если приложение не имеет доступа к данному свойству, генерируется исключение.

Класс Preferences

Как видим, класс `Properties` позволяет легко загружать и сохранять информацию о конфигурации системы. Однако файлы свойств обладают рядом недостатков.

- Конфигурационные файлы не всегда хранятся там же, где и программа, и не всегда доступны для записи. Например, файл может находиться в каталоге, предназначенном только для чтения, или в JAR-файле.
- Разные пользователи могут по-разному настраивать одно и то же приложение.
- В ряде случаев невозможно хранить конфигурационные файлы в рабочем каталоге пользователя. Некоторые операционные системы (например, Windows 9x) вообще не поддерживают рабочие каталоги.
- Стандартные соглашения об именах конфигурационных файлов отсутствуют. Таким образом, если пользователь инсталлирует несколько приложений, возникает опасность конфликта имен.

Некоторые операционные системы хранят информацию о своих установках в центральном хранилище конфигурационных данных. Наиболее известным примером такого хранилища является системный реестр Windows. Класс `Preferences`, реализованный в JDK 1.4, поддерживает подобное центральное хранилище, не зависящее от операционной системы. На платформе Windows класс `Preferences` использует системный реестр, на платформе Linux информация хранится в локальном системном файле. Разумеется, реализация хранилища абсолютно прозрачна для программиста, использующего класс `Preferences`.

Хранилище `Preferences` имеет древовидную структуру, в узлах которой содержатся имена путей, например `/com/mycompany/myapp`. Как и в случае пакетов, рекомендуется во избежание конфликта имен формировать имена на основе доменных имен, записывая их в обратном порядке. Разработчики интерфейса API предполагают, что пути к узлам совпадают с именами пакетов, используемых в программе.

Каждый узел в хранилище содержит отдельную таблицу пар ключ–значение, которую можно использовать для записи чисел, строк или байтовых массивов. Возможность хранить сериализованные объекты не предусмотрена. Разработчики интерфейса API чувствовали, что формат сериализации слишком хрупок и обеспечить его долговременное хранение непросто. Разумеется, если вы не согласны с этим, можно хранить сериализованные объекты в виде байтовых массивов.

Для обеспечения дополнительной степени гибкости предусмотрено несколько параллельных деревьев. Каждая программа использует одно дерево. Кроме того, существует дополнительное дерево, называемое системным и предназначенное для хранения настроек, общих для все пользователей. Для доступа к соответствующему пользовательскому дереву класс `Preferences` использует понятие "текущий пользователь".

Поиск требуемого узла начинается с пользовательского или системного корня.

```
Preferences root = Preferences.userRoot();
```

или

```
Preferences root = Preferences.systemRoot();
```

Теперь для доступа к узлу достаточно указать соответствующий путь.

```
Preferences node = root.node("/com/mycompany/myapp");
```

Существует удобный способ получения доступа к узлу, путь к которому совпадает с именем пакета, содержащего класс. Для этого нужно взять объект этого класса и вызвать один из следующих методов:

```
Preferences node = Preferences.userNodeForPackage(obj.getClass());
```

или

```
Preferences node = Preferences.systemNodeForPackage(obj.getClass());
```

Как правило, параметр `obj` является ссылкой `this`.

Получив доступ к узлу, можно обратиться к таблице ключ–значение, используя указанные ниже методы.

```
String get(String key, String defval)
int getInt(String key, int defval)
long getLong(String key, long defval)
float getFloat(String key, float defval)
double getDouble(String key, double defval)
boolean getBoolean(String key, boolean defval)
byte[] getByteArray(String key, byte[] defval)
```

Обратите внимание на то, что если хранилище может оказаться недоступным, то при считывании этой информации необходимо указывать значения, предусмотренные по умолчанию. Эти значения необходимы по нескольким причинам. Во-первых, *данные могут быть неполными, так как пользователи не всегда указывают все параметры*. Во-вторых, на некоторых платформах при недостатке ресурсов хранилища не могут создаваться. В-третьих, переносные устройства могут быть временно отсоединены от хранилища.

С помощью методов `put` можно записать данные в хранилище.

```
put(String key, String value)
putInt(String key, int value)
```

и т.д.

Центральным хранилищам, например системному реестру Windows, традиционно присущи два недостатка.

- В них постепенно накапливается устаревшая информация.
- *Данные о конфигурации системы становятся все более запутанными*, что затрудняет перенос установок на новую платформу.

Класс `Preferences` *позволяет решить вторую проблему. Данные можно экспортировать* в поддерево (или, в некоторых случаях, в отдельный узел), вызвав один из приведенных ниже методов.

```
void exportSubtree(outputStream out)
void exportNode(outputStream out)
```

Информация хранится в XML-формате. Ее можно импортировать в другое хранилище, вызвав следующий метод:

```
void importPreferences(inputStream in)
```

Рассмотрим пример XML-файла, содержащего конфигурационные данные.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE preferences SYSTEM
   "http://java.sun.com/dtd/preferences.dtd">
<preferences EXTERNAL_XML_VERSION="1.0">
  <root type="user">
    <map/>
    <node name="com">
      <map/>
      <node name="horstmann">
        <map/>
        <node name="corejava">
          <map>
            <entry key="left" value="11"/>
            <entry key="top" value="9"/>
            <entry key="width" value="453"/>
            <entry key="height" value="365"/>
          </map>
        </node>
      </node>
    </node>
  </root>
</preferences>
```

Если ваша программа использует параметры настройки, можно предоставить ее пользователям возможность экспортировать и импортировать их, что упрощает перенос программы с одного компьютера на другой. Код, приведенный в листинге 10.16, иллюстрирует данный подход. Программа сохраняет координаты и размер главного окна. Попробуйте изменить размеры окна, затем завершите приложение и запустите его снова. Размеры окна должны совпадать с размерами, указанными перед окончанием работы программы.

Листинг 10.16. Содержимое файла `PreferencesTest.java`

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.logging.*;
import java.util.prefs.*;
import javax.swing.*;
import javax.swing.event.*;

/**
   Программа тестирования средств настройки.
   Запоминает координаты и размер фрейма.
*/
public class PreferencesTest
{
   public static void main(String[] args)
   {
      PreferencesFrame frame = new PreferencesFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

```
/**
   Фрейм восстанавливает координаты и размер, заданные
   пользователем, и сохраняет установки перед выходом
   из программы.
*/
class PreferencesFrame extends JFrame
{
   public PreferencesFrame()
   {
      setTitle("PreferencesTest");

      // Определение координат и размеров на основе
      // сохраненных установок.

      Preferences root = Preferences.userRoot();
      final Preferences node = root.node("/com/horstmann/corejava");
      int left = node.getInt("left", 0);
      int top = node.getInt("top", 0);
      int width = node.getInt("width", DEFAULT_WIDTH);
      int height = node.getInt("height", DEFAULT_HEIGHT);
      setBounds(left, top, width, height);

      // Настройка диалогового окна для выбора XML-файлов.

      final JFileChooser chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File("."));

      // Допускаются файлы, имена которых заканчиваются .xml.
      chooser.setFileFilter(new
         javax.swing.filechooser.FileFilter()
         {
            public boolean accept(File f)
            {
               return f.getName().toLowerCase().endsWith(".xml")
                     || f.isDirectory();
            }
            public String getDescription()
            {
               return "XML files";
            }
         });


      // Настройка меню.
      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);
      JMenu menu = new JMenu("File");
      menuBar.add(menu);

      JMenuItem exportItem = new JMenuItem("Export preferences");
      menu.add(exportItem);
      exportItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
```

```
            if(chooser.showSaveDialog(PreferencesFrame.this) ==
                JFileChooser.APPROVE_OPTION)
            {
               try
               {
                  OutputStream out = new
                     FileOutputStream
                     (chooser.getSelectedFile());
                  node.exportSubtree(out);
                  out.close();
               }
               catch (Exception e)
               {
                  e.printStackTrace();
               }
            }
         }
      });

      JMenuItem importItem = new JMenuItem("Import preferences");
      menu.add(importItem);
      importItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               if(chooser.showOpenDialog(PreferencesFrame.this) ==
                  JFileChooser.APPROVE_OPTION)
               {
                  try
                  {
                     InputStream in = new
                       FileInputStream(chooser.getSelectedFile());
                     node.importPreferences(in);
                     in.close();
                  }
                  catch (Exception e)
                  {
                     e.printStackTrace();
                  }
               }
            }
         });

      JMenuItem exitItem = new JMenuItem("Exit");
      menu.add(exitItem);
      exitItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               node.putInt("left", getX());
               node.putInt("top", getY());
               node.putInt("width", getWidth());
               node.putInt("height", getHeight());
               System.exit(0);
```

```
            }
        });
    }
    public static final int DEFAULT_WIDTH = 300;
    public static final int DEFAULT_HEIGHT = 200;
}
```

java.util.prefs.Preferences 1.4

- `Preferences userRoot()`

  Возвращает из дерева настроек вызывающей программы корневой узел для пользователя.

- `Preference systemRoot()`

  Возвращает системный корневой узел из дерева настроек.

- `Preferences node(String path)`

  Возвращает узел, достижимый из текущего узла с помощью заданного пути. Если параметр `path` задает абсолютный путь (т.е. начинается с косой черты /), то узел достигается из корня дерева. Если узел отсутствует, он создается.

- `Preferences userNodeForPackage(Class c1)`
- `Preferences systemNodeForPackage(Class c1)`

  Возвращает узел из дерева текущего пользователя или системного дерева, абсолютный путь к которому соответствует имени пакета, содержащего класс `c1`.

- `String[] keys()`

  Возвращает все ключи, принадлежащие данному узлу.

- `String get(String key, String defval)`
- `int getInt(String key, int defval)`
- `long getLong(String key, long defval)`
- `float getFloat(String key, float defval)`
- `double getDouble(String key, double defval)`
- `boolean getBoolean(String key, boolean defval)`
- `byte[] getByteArray(String key, byte[] defval)`

  Эти методы возвращают значение, связанное с указанным ключом. Если значения в таблице нет, либо оно имеет неверный тип, либо сама таблица недоступна, возвращается значение, предусмотренное по умолчанию.

- `void put(String key, String value)`
- `void putInt(String key, int value)`
- `void putLong(String key, long value)`
- `void putFloat(String key, float value)`
- `void putDouble(String key, double value)`
- `void putBoolean(String key, boolean value)`
- `void putByteArray(String key, byte[] value)`

  Эти методы записывают пару ключ–значение в заданный узел.

- `void exportSubtree(OutputStream out)`

Выводит в указанный поток параметры настройки, хранящиеся в заданном узле и узлах, дочерних по отношению к нему.

- `void exportNode(OutputStream out)`

  Выводит в указанный поток параметры настройки, хранящиеся в заданном узле (игнорируя дочерние узлы).

- `void importPreferences(InputStream in)`

  Импортирует параметры настройки, содержащиеся в указанном потоке.

На этом мы завершаем обсуждение способов доставки программного обеспечения, написанного на языке Java. В следующей главе описана обработка исключений, возникающих во время работы программы. Кроме того, мы дадим несколько советов, касающихся способов тестирования и отладки программ и позволяющих избежать многих проблем в ходе их выполнения.

Глава 11

Исключения и отладка

В этой главе...

- Обработка ошибок
- Перехват исключений
- Как использовать исключения
- Регистрация ошибок
- Диагностические утверждения
- Приемы отладки
- Применение отладчика

В идеальном мире пользователи никогда не делают ошибок при вводе данных, файлы, которые они пытаются открыть, всегда существуют, а программы всегда работают правильно. До сих пор мы предполагали, что живем именно в таком мире. Теперь пришла пора спуститься на землю, населенную реальными людьми, делающими ошибки при программировании и вводе данных, и рассмотреть механизмы языка Java, которые позволяют в этом мире как-то существовать.

Обнаруживать ошибки всегда неприятно. Если пользователь однажды потеряет все результаты своей работы из-за скрытой ошибки или непредвиденного стечения обстоятельств, он может навсегда отвернуться от такой программы. В подобной ситуации необходимо сделать, по крайней мере, следующее.

- Сообщить пользователю об обнаруженной ошибке.
- Сохранить все результаты его работы.
- Дать ему возможность аккуратно завершить программу.

В нештатных ситуациях, например при некорректном вводе данных, которые могут привести к разрушению программы, язык Java использует механизм исправления ошибок, который называется *обработкой исключений* (exception handling). Этот механизм аналогичен обработке исключений в языках C++ и Delphi. Он описывается в первой части этой главы.

Вторая часть главы посвящена способам обнаружения ошибок до того, как они приведут к возникновению исключений во время выполнения программы. К сожалению, средства обнаружения ошибок, предусмотренные в JDK, существенно устарели. Для того чтобы как-то облегчить вашу работу, мы дадим несколько советов. Затем мы объясним, как использовать отладчик, запускаемый в режиме командной строки. Для разработки серьезных программ на языке Java следует воспользоваться средствами отладки, предусмотренными в интегрированных средах Eclipse, NetBeans и JBuilder. В данной главе описывается отладчик среды Eclipse.

Обработка ошибок

Предположим, что в ходе выполнения программы, написанной на языке Java, обнаруживается ошибка. Она может быть вызвана неверной информацией в файле, неправильным сетевым соединением или выходом за пределы допустимого диапазона изменения индексов массива, а может быть попыткой сослаться на несуществующий объект. Вполне естественно, пользователи надеются, что программа сама справится с возникшими неприятностями. Если из-за ошибки какая-либо операция не может быть завершена, то программа должна сделать одно из двух.

- Вернуться в безопасное состояние и позволить пользователю выполнить другие команды.
- Дать пользователю возможность сохранить результаты своей работы и аккуратно завершить работу.

Это может оказаться нелегко: фрагмент кода, в котором обнаруживается ошибка (или даже тот, выполнение которого становится причиной ошибки), обычно располагается довольно далеко от кода, который может восстановить данные и сохранить результаты, полученные пользователем. Механизм обработки исключений предназначен для передачи данных обработчику ошибок из того места, где возник сбой. Предусматривая обработку исключений в программе, нужно предвидеть возможные ошибки и проблемы, связанные с ними. Какие же ошибки следует рассмотреть в первую очередь?

Ошибки ввода. Пользователи часто делают опечатки. Допустим, пользователь хочет соединиться с Web-узлом, но допустил синтаксическую ошибку, задавая URL. Ваша программа должна была бы проверить синтаксис, но, предположим, вы забыли это сделать. Тогда сетевое программное обеспечение сообщит об ошибке.

Сбои оборудования. Аппаратное обеспечение не всегда работает так, как вы ожидаете. Принтер может оказаться выключенным, а Web-страница — временно недоступной. Оборудование часто отказывает в процессе работы, например, принтер может отказать на середине страницы.

Физические ограничения. Диск может оказаться переполненным, а оперативная память — исчерпанной.

Ошибки программирования. Какой-либо метод может работать неправильно. Например, он может возвращать неверный результат или некорректно вызывать другие методы. Выход за пределы массива, попытка найти несуществующий элемент таблицы хэширования, извлечение элемента из пустого стека — все это примеры ошибок программирования.

Обычно метод сообщает об ошибке, возвращая специальный код, который анализируется вызывающим методом. Например, методы, считывающие информацию из файлов, по достижении конца файла обычно возвращают значение –1. Такой способ обработки ошибок часто оказывается эффективным. В других случаях в качестве признака ошибки возвращается значение `null`. В главе 10 мы рассмотрели пример, в котором метод `getParameter()` класса `Applet` возвращал `null`, если требуемый параметр отсутствовал.

К сожалению, возвращать код ошибки не всегда возможно. Иногда не удается различить правильные данные и признак ошибки. Метод, возвращающий целое число, не может, обнаружив ошибку, вернуть значение –1, поскольку оно может быть результатом вычислений.

Как мы упоминали в главе 5, в языке Java есть возможность предусмотреть в каждом методе альтернативный выход, которым следует воспользоваться, если задачу невозможно выполнить до конца. При этом метод не станет возвращать значение, а *сгенерирует* (throw) *объект, инкапсулирующий информацию об ошибке. Обратите внимание* на то, что выход из метода происходит незамедлительно и он не возвращает своего нормального значения. Более того, возобновить выполнение кода, вызвавшего данный метод, невозможно. Вместо этого начинается поиск *обработчика исключения*, который может справиться с возникшей ошибкой.

Исключения имеют свой собственный синтаксис и являются частью особой иерархии наследования. Сначала мы рассмотрим вопросы, связанные с синтаксисом, а затем дадим несколько советов по эффективному использованию обработчиков исключений.

Классификация исключений

В языке Java объект исключения всегда является экземпляром класса, производного от класса `Throwable`. Как мы вскоре убедимся, если стандартных классов недостаточно, можно создавать и свои собственные классы исключений.

На *рис. 11.1 упрощенно показана иерархия исключений в языке* Java.

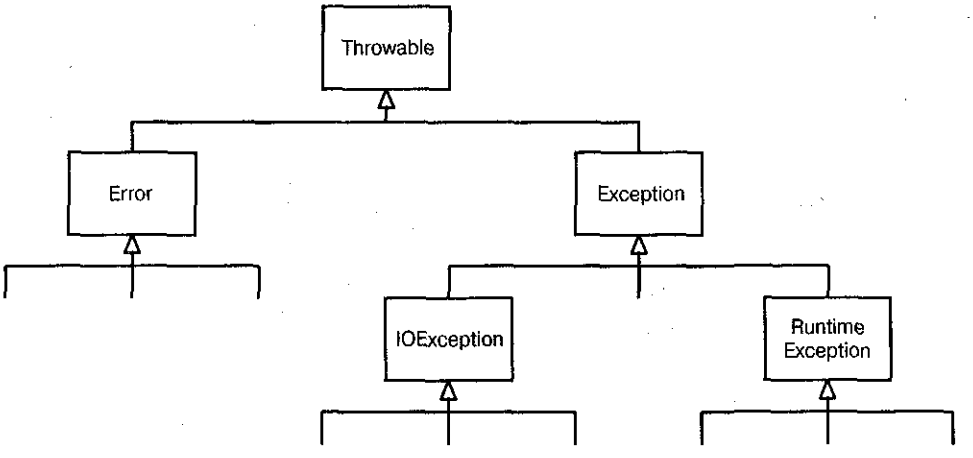


Рис. 11.1. Иерархия исключений в языке Java

Обратите внимание на то, что, хотя все исключения имеют общего предка `Throwable`, иерархия сразу разделяется на две ветви: `Error` и `Exception`.

Иерархия `Error` описывает внутренние ошибки и ситуации, связанные с нехваткой ресурсов, в системе поддержки выполнения программ. Ни один объект этого типа самостоятельно сгенерировать невозможно. При возникновении внутренней ошибки возможности программиста крайне ограничены.

Программируя на языке Java, основное внимание следует уделять иерархии `Exception`. Эта иерархия также разделяется на две ветви: исключения, производные от класса `RuntimeException`, и остальные. Исключения типа `RuntimeException` возникают вследствие ошибок программирования. Все другие исключения являются следствием непредвиденного стечения обстоятельств, например ошибок ввода-вывода, возникающих при выполнении вполне корректных программ.

Исключения, производные от класса `RuntimeException`, связаны со следующими проблемами.

- Неверное приведение типов.
- Выход за пределы массива.
- Попытка обратиться к объекту по ссылке `null`.

Остальные исключения возникают в следующих случаях.

- Попытка чтения после конца файла.
- Попытка установить соединение, используя неправильный URL.
- Попытка получить объект `Class`, если в строке указан несуществующий класс.

Исключения `RuntimeException` практически всегда возникают по вине программиста. Исключения `ArrayIndexOutBoundsException` можно избежать, если всегда проверять индексы массива. Исключение `NullPointerException` никогда не возникнет, если перед использованием переменной вы проверите, не содержит ли она значение `null`.

А как насчет неверного URL-адреса? Всегда ли можно обнаружить ошибку заранее? Разные браузеры обрабатывают разные URL-адреса. Например, браузер `Netscape` может работать с адресами вида `mailto:URL`, а программа `appletviewer` — нет. Следовательно, термин "неправильный" (malformed) зависит от среды, а не от программы.

В спецификации языка Java любое исключение, производное от класса `Error` или `RuntimeError`, называется *неконтролируемым* (unchecked). Все остальные исключения называются *контролируемыми* (checked). Для всех контролируемых исключений компилятор проверяет наличие соответствующих обработчиков.

Имя `RuntimeException` иногда вводит в заблуждение. Разумеется, все ошибки, которые мы обсуждаем, возникают в ходе выполнения программы.

Если вы знакомы с иерархией исключений из стандартной библиотеки языка C++, то, возможно, заметили некоторое противоречие в именах. В языке C++ есть два основных класса, описывающих исключения: `runtime_error` и `logic_error`. Класс `logic_error` эквивалентен классу `RuntimeException` языка Java. Класс `runtime_error` является базовым для исключений, вызванных непредвиденным стечением обстоятельств. Он является эквивалентом всех исключений, предусмотренных в языке Java, не относящихся к типу `RuntimeException`.

Объявление контролируемых исключений

Метод может генерировать исключения, если возникают обстоятельства, с которыми он не справляется. Компилятору метод не только сообщает, какие значения он может возвращать, но и предсказывает, *какие ошибки могут возникнуть*. Например, код, считывающий информацию из некоего файла, может предположить, что такого файла не существует или он пуст. Следовательно, код, предназначенный для чтения из файла, должен сообщить компилятору, что он может генерировать исключение `IOException`.

Объявление о том, что данный метод может генерировать исключение, помещается в его заголовок. Ниже в качестве примера приведен заголовок метода, принадлежащего классу `BufferedReader` из стандартной библиотеки. Этот метод считывает текстовую строку из входного потока, связанного, например, с файлом или сетевым соединением. (Потоки рассматриваются в главе 12.)

```
public String readLine() throws IOException
```

В заголовке указано, что данный метод возвращает строку, но, кроме того, при возникновении ошибок может генерировать исключение `IOException`. Это означает, что в случае ошибки метод не вернет строку, а сгенерирует объект класса `IOException`. Если это произойдет, система поддержки выполнения программ начнет поиск обработчика, предназначенного для работы с объектами класса `IOException`.

Создавая свой собственный метод, не нужно объявлять все возможные исключения, который ваш метод фактически может генерировать. Чтобы понять, когда (и что) следует описывать с помощью выражения `throws`, имейте в виду, что исключение генерируется в следующих четырех случаях.

1. Вызывается метод, генерирующий контролируемое исключение, например метод `readLine()` класса `BufferedReader`.
2. Обнаружена ошибка, и с помощью оператора `throw` явным образом создается контролируемое исключение (оператор `throw` обсуждается далее в этой главе).
3. Обнаружена ошибка программирования, например, в программе присутствует выражение `a[-1] = 0`, вследствие чего возникает неконтролируемое исключение, например `ArrayIndexOutOfBoundsException`.
4. Возникла внутренняя ошибка в виртуальной машине или оперативной библиотеке.

В первых двух случаях нужно сообщить программистам, которые будут использовать ваш метод, что возможно исключение. Зачем? Любой метод, генерирующий исключение, представляет собой смертельную угрозу. Если в программе не предусмотреть обработку исключения, то выполнение текущего потока прерывается.

Объявить, что метод может генерировать исключение, можно, включая в его заголовок выражение `throws`.

```
class MyAnimation
{
   ...
   public Image loadImage(String s) throws IOException
   {
      ...
   }
}
```

Если метод может генерировать несколько контролируемых исключений, всех их нужно перечислить в его заголовке, разделяя запятыми.

```
class MyAnimation
{
   ...
   public Image loadImage(String s)
      throws EOFException, MalformedURLException
   {
      ...
   }
}
```

Однако внутренние ошибки, т.е. исключения, производные от класса `Error`, объявлять не нужно. Эти ситуации могут генерироваться любыми методами, причем повлиять на них невозможно.

Точно так же не нужно объявлять неконтролируемые исключения, производные от класса `RuntimeException`.

```
class MyAnimation
{
   ...
   void drawImage(int i)
      throws ArrayIndexOutOfBoundsException // Плохой стиль!
   {
      ...
   }
}
```

Эти ошибки лежат полностью на вашей совести. Если вас так беспокоит возможность выхода индекса массива за пределы диапазона, лучше уделите больше внимания ее предотвращению, а не заявляйте, что она может возникнуть.

Отметим также, что метод должен объявлять все контролируемые исключения, которые он может генерировать. Неконтролируемые исключения либо находятся вне вашей компетенции (класс `Error`), либо являются следствием логических ошибок, которые не следовало допускать (класс `RuntimeException`). Если ваш метод не сообщает обо всех контролируемых исключениях, компилятор выдаст сообщение об ошибке.

Разумеется, как мы уже видели, вместо объявления исключений их можно перехватывать. В этом случае исключение не генерируется и спецификация `throws` не нужна. Позднее в этой главе мы рассмотрим вопрос, что лучше: перехватывать исключение самому или поручить это кому-нибудь другому.

Метод подкласса не может генерировать более общие исключения, чем замещаемый им метод суперкласса. (Исключения могут только конкретизироваться или не возникать вовсе.) В частности, если метод суперкласса вообще не генерирует контролируемые исключения, то и подкласс этого сделать не может. Например, если вы переопределяете метод `JComponent.paintComponent()`, ваш метод `paintComponent()` не должен генерировать ни одного контролируемого исключения, поскольку соответствующий метод суперкласса этого не делает.

Если метод объявляет, что может генерировать исключение определенного класса, то он может генерировать также исключения его подклассов. Например, метод `readLine()` класса `BufferedReader` объявляет, что он может генерировать исклю-

чение класса IOException, причем нам неизвестно, какое именно. Это может быть ситуация как самого класса IOException, так и его потомков, например класса EOFException.

Спецификатор throws совпадает со спецификатором throws языка C++. Однако существует одно важное отличие. В языке C++ throws действует только во время выполнения программы, а не на этапе компиляции. Это значит, что компилятор языка C++ игнорирует объявление исключений. Однако если исключение генерируется в функции, не являющейся частью списка throw, то вызывается функция unexpected и по умолчанию выполнение программы прекращается. Кроме того, в языке C++, если выражеие throw не указано, функция может генерировать любое исключение.

Генерация исключений

Предположим, что с вашей программой случилось нечто ужасное. У вас есть метод readData(), считывающий данные из файла, в заголовке которого указано следующее.

```
Content-length: 1024
```

Однако после ввода 733 символов файл окончился. Вы считаете эту ситуацию настолько ненормальной, что решаете генерировать исключение.

Теперь нужно решить, какого типа должно быть исключение. Нам подходит одна из разновидностей класса IOException. Просматривая документацию, вы обнаруживаете описание исключения EOFException: "Сигнализирует о том, что во время ввода данных неожиданно обнаружен признак конца файла EOF". Отлично. Это исключение можно создать двумя способами:

```
throw new EOFException();
```

или, что то же самое,

```
EOFException e = new EOFException();
throw e;
```

Итак, для решения задачи нужно написать следующий фрагмент кода:

```
String readData(Scanner in) throws EOFException
{
   . . .
   while (. . .)
   {
      if (!in.hasNext()) // Обнаружен EOF.
      {
         if (n < len)
            throw new EOFException();
      }
      . . .
   }
   return s;
}
```

В классе EOFException есть второй конструктор, получающий в качестве параметра строку. Его можно использовать для более подробного описания исключения.

```
String gripe = "Content-length: " + len ", Received: + n;
throw new EOFException(gripe);
```

Задача облегчается, если один из существующих классов исключений подходит для вашей цели. В этом случае нужно сделать следующее.

1. Найти подходящий класс.
2. Создать экземпляр этого класса.
3. Генерировать исключенрие.

После генерации исключения управление уже не будет возвращено вызывающему методу. Это означает, что беспокоиться о возвращении значения, предусмотренного по умолчанию, или кода ошибки не нужно.

Генерация исключений в языках C++ и Java происходит почти одинаково; есть лишь одно небольшое различие. В языке Java можно генерировать только объекты классов, производных от класса `Throwable`, а в языке C++ можно генерировать объекты любого типа.

Создание классов исключений

В программе может возникнуть ситуация, не предусмотренная ни одним из существующих классов. В этом случае можно достаточно просто создать свой собственный класс исключения. Очевидно, что он должен быть подклассом класса `Exception` или одного из его дочерних классов, например `IOException`. Этот класс можно снабдить конструктором по умолчанию и конструктором, содержащим подробное сообщение. (Это сообщение может быть полезно при отладке. При его выводе используется метод `toString()` класса `Throwable`.)

```
class FileFormatException extends IOException
{
   public FileFormatException() {}
   public FileFormatException(String gripe) {}
   {
      super(gripe);
   }
}
```

Теперь мы готовы генерировать исключение своего собственного типа.

```
String readData(BufferReader in) throws FileFormatException
{
   ...
   while (...)
   {
      if (ch == -1) // Обнаружен маркер конца файла EOF.
      {
         if ( n < len)
            throw new FileFormatException();
      }
      ...
   }
   return s;
}
```

API **java.lang.Throwable 1.0**

- `Throwable()`

  Создает новый объект класса `Throwable`, не сопровождая его подробным сообщением.

- `Throwable(String message)`

  Создает новый объект класса `Throwable`, сопровождая его подробным сообщением. По соглашению все классы исключений должны содержать оба конструктора.

- `String getMessage()`

  Получает подробное сообщение, предусмотренное в объекте `Throwable`.

Перехват исключений

Теперь мы знаем, как генерировать исключения. Ничего сложного в этом нет: сгенерировав исключение, о нем можно попросту забыть. Разумеется, должен существовать код, позволяющий обрабатывать его.

Если исключение возникает и нигде не перехватывается, то программа прекращает работу, выводя на консоль сообщение о типе исключения и содержимое стека. Графические программы (аплеты и приложения) выводят то же сообщение, продолжая при этом поддерживать интерфейс пользователя. (Отлаживая графические программы, лучше держать консольное окно на экране, не минимизируя его.)

Перехват исключения осуществляется с помощью блока `try/catch`. В простейшем случае этот блок имеет следующий вид:

```
try
{
   Код.
}

catch (тип_исключения e)
{
   Обработчик исключения данного типа.
}
```

Если фрагмент кода, содержащийся в блоке `try`, генерирует исключение типа, указанного в заголовке блока `catch`, то происходит следующее.

1. Программа пропускает оставшуюся часть кода в блоке `try`.
2. Программа выполняет код обработчика, указанный в блоке `catch`.

Если код в блоке `try` не генерирует исключение, то программа игнорирует блок `catch`.

Если какой-либо из операторов блока `try` генерирует исключение, отличное от типа указанного в выражении `catch`, выполнение метода немедленно прекращается. (Будем надеяться, что в вызывающем методе перехват исключения все же предусмотрен.)

Рассмотрим следующий пример:

```
public void read(String filename)
{
   try
   {
      InputStream in = new FileInputStream(filename);
      int b;
      while ((b = in.read()) !=-1)
      {
         обработка
      }
   }
   catch (IOException exception)
   {
      exception.printStackTrace();
   }
}
```

Обратите внимание на то, что большая часть кода в блоке `try` выполняет простейшие действия: считывает и обрабатывает строки, пока не обнаружится конец файла. Как указано в документации, существует возможность того, что метод `read()` сгенерирует исключение `IOException`. В данном случае мы пропускаем весь цикл `while`, переходим к разделу `catch` и осуществляем трассировку стека. Для небольших программ это вполне разумно. А какие еще существуют возможности?

Зачастую лучше вовсе ничего не делать. Если в методе `read()` возникла ошибка, возложите ответственность за обработку этой ситуации на вызывающий метод! Придерживаясь этого подхода, достаточно указать, что метод `read()` может генерировать исключение `IOException`.

```
public void read(String filename) throws IOException
{
   InputStream in = new FileInputStream(filename);
   int b;
   while ((b = in.read()) !=-1)
   {
      process input
   }
}
```

Запомните, компилятор строго следит за спецификаторами `throws`. Вызывая метод, генерирующий контролируемое исключение, нужно либо самому обрабатывать его, либо делегировать обработку другому методу.

Что лучше? Как правило, следует перехватывать лишь те исключения, которые вы сами можете обработать, а остальные передавать дальше. Передавая исключение, нужно добавлять спецификатор `throws`, чтобы предупредить об этом вызывающий метод.

Просмотрите документацию по API и обратите внимание на методы, которые могут генерировать исключения. Затем примите решение, обрабатывать исключение или включить в список `throws`. Во втором варианте нет ничего постыдного — лучше предоставить обработку исключения более компетентному обработчику.

У этого правила есть одно исключение. Если вы создаете метод, который переопределяет метод суперкласса, не генерирующий ни одного исключения (например, метод `paintComponent` класса `JComponent`), вы *обязаны* сами перехватывать каждое

контролируемое исключение в теле метода. В этот метод подкласса нельзя добавить спецификатор `throws`, отсутствующий в методе суперкласса.

Перехват исключений в языках Java и C++ осуществляется почти одинаково. Выражение

```
catch (Exception e) // Java
```

является аналогом выражения

```
catch (Exception& e) // C++
```

Перехват нескольких исключений

С помощью блока `try` можно перехватить несколько исключений, обработав их по отдельности. Для каждого типа исключения следует предусмотреть свой блок `catch`.

```
try
{
   Код, который может генерировать исключения.
}
catch (MalformedURLException e1)
{
   // Действия в случае некорректного формата URL.
}
catch (UnknownHostException e2)
{
   // Действия в случае, если URL не существует.
}

catch (IOException e3)
{
   // Действия при возникновении ошибок ввода-вывода.
}
```

Объект исключения (`e1`, `e2` или `e3`) содержит информацию о нем. Для того чтобы узнать сведения о сообщении, нужно вызвать следующий метод:

```
e3.getMessage()
```

Фактический тип объекта исключения можно определить с помощью метода

```
e3.getClass().getName()
```

Повторная генерация исключения

Исключение можно генерировать в блоке `catch`, создавая тем самым цепочку исключений. Обычно разработчики поступают так в том случае, если им надо изменить тип исключения. Если вы создаете подсистему, используемую другими программистами, имеет смысл генерировать такие исключения, которые дали бы возможность сразу определить, что ошибка возникла именно в ней. В качестве примера подобного исключения можно привести `ServletException`. Возможно, что контейнеру сервлетов и не обязательно иметь информацию о том, какая именно возникла ошибка, а важно лишь знать, что некорректно работает сервлет.

Ниже показан пример перехвата исключения и повторной его генерации.

```
try
{
   Обращение к базе данных
}
catch (SQLException e)
{
   throw new ServletException("database error: " +
      e.getMessage());
}
```

В данном случае в конструкторе `ServletException` формируется сообщение. В JDK 1.4 вы можете даже указать исходное исключение, сообщив, что оно стало причиной нового.

```
try
{
   Обращение к базе данных
}
catch (SQLException e)
{
   Throwable se = new ServletException("database error");
   se.setCause(e);
   throw se;
}
```

Теперь при перехвате исключения можно извлечь исходное исключение так, как показано ниже.

```
Throwable e = se.getCause();
```

Специалисты настоятельно рекомендуют использовать при разработке именно такой подход. Он позволяет генерировать исключения более высокого уровня, не теряя деталей исходного исключения.

Описанный подход удобен в том случае, если перехват исключения осуществляется в методе, который не имеет права генерировать контролируемые исключения. Вы можете перехватить контролируемое исключение и инкапсулировать его в составе неконтролируемого.

В ряде классов, описывающих исключения, например в `ClassNotFoundException`, `InvocationTargetException` и `RuntimeException`, поддерживается своя схема создания цепочки исключений. Начиная с версии JDK 1.4, в них параллельно поддерживается стандартный механизм инкапсуляции исходного исключения. Таким образом, вы можете либо применять методы, типичные для этих классов, либо использовать метод `getCause()`. К сожалению, данное новшество не распространяется на класс `SQLException`, очень часто применяемый при разработке приложений.

Блок finally

Когда код генерирует исключение, операторы, оставшиеся до конца метода, не выполняются. Если метод задействовал какие-то локальные ресурсы, о которых известно лишь ему, то освободить их окажется невозможно. Можно, конечно, повторно генерировать все исключения, однако это решение не слишком удачно, поскольку ресурсы нужно освобождать в двух местах: в нормальном коде и в коде исключения.

В языке Java есть лучшее решение — использовать блок `finally`. Ниже приведен пример корректного вызова метода `dispose()` объекта `Graphics`. Если вы работаете с базами данных, то можете использовать тот же подход для того, чтобы правильно закрыть соединение с базой. Как вы увидите в главе 4 второго тома, закрывать соединение с базой необходимо даже в том случае, если при выполнении программы возникло исключение.

Код в блоке `finally` выполняется независимо от того, возникло исключение или нет. В частности, в приведенном ниже примере метод `dispose()` выполняется при любых условиях.

```
Graphics g = image.getGraphics();
try
{
   // 1
   Код, который может генерировать исключения.
   // 2
}
catch (IOException e)
{
   // 3
   Выводим диалоговое окно.
   // 4
}
finally
{
   // 5
   g.dispose();
}
// 6
```

Рассмотрим три возможные ситуации, в которых программа выполняет раздел `finally`.

1. Код не генерирует никаких исключений. В этом случае программа сначала полностью выполняет блок `try`, а затем блок `finalize`. Иными словами, при работе программы последовательно выполняются этапы 1, 2, 5 и 6.
2. Код генерирует исключение, которое перехватывается в разделе `catch` (в данном примере это исключение класса `IOException`). В этом случае программа сначала выполняет блок `try` до той точки, в которой возникает исключение. Остальная часть блока `try` игнорируется. Затем программа выполняет код из соответствующего блока `catch` и в заключение — блок `finally`.

   Если в разделе `catch` исключения не генерируются, то программа выполняет первую строку вслед за блоком `try`. Таким образом, при описанной ситуации выполняются этапы 1, 3, 4, 5 и 6.

   Если исключение генерируется в блоке `catch`, то управление передается вызывающему методу и выполняются лишь этапы 1, 3 и 5.
3. Код генерирует исключение, которое не обрабатывается `catch`. В этом случае программа выполняет блок `try` вплоть до той точки, в которой генерируется исключение. Оставшаяся часть блока `try` игнорируется. Затем программа выполняет код из блока `finally` и передает исключение обратно в вызывающий метод.

Блок finally можно использовать без блока catch. Рассмотрим следующий пример:

```
Graphics g = image.getGraphics();
try
{
{
   Код, который может генерировать исключения.
}
finally
{
   g.dispose();
}
```

Метод g.dispose() в блоке finally выполняется независимо от того, возникает ли какое-либо исключение в блоке try. Разумеется, если исключение возникает, она будет перехвачено другим разделом catch.

Мы настоятельно советуем вам разделять блоки try/catch и try/finally. При этом код программы становится более понятным. Рассмотрим следующий пример:

```
InputStream in = ...;
try
{
   try
   {
      Код, который может генерировать исключение.
   }
   finally
   {
      in.close();
   }
}
catch (IOException e)
{
   Отображение сообщения об ошибке.
}
```

Здесь внутренний блок try отвечает только за закрытие входного потока. Внешний блок try сообщает об ошибках. Такой код не только более понятен, но и более функционален, так как ошибки в блоке finally также распознаются.

Если в блоке finally присутствует оператор return, результаты его выполнения могут быть неожиданными. Предположим, что в середине блока try вы вышли из метода с помощью оператора return. Перед тем как передать управление вызывающему методу, должен быть выполнен блок finally. Если он тоже содержит return, то возвращаемое значение будет замаскировано. Рассмотрим пример:

```
public static int f(int n)
{
   try
   {
      int r = n * n;
      return r;
   }
```

```
        finally
        {
           if (n == 2) return 0;
        }
    }
```

Если выполнить вызов `f(2)`, то блок `try` вычислит значение `r = 4` и выполнит оператор `return`. Однако на самом деле до оператора `return` будет выполнен код раздела `finally`. Этот раздел заставит метод вернуть значение 0, игнорируя исходное значение, равное 4.

Иногда раздел `finally` повергает программистов в уныние, в частности, если методы очистки памяти могут генерировать исключения. Типичный случай — закрытие потока (подробнее об этом — в главе 12). Допустим, при возникновении исключения мы хотим закрыть поток.

```
InputStream in;
try
{
   Код, который может генерировать исключения.
}
catch (IOException e)
{
   Отображение диалогового окна.
}
finally
{
   in.close();
}
```

Предположим также, что код в блоке `try` генерирует некоторое исключение, *отличающееся от класса* `IOException`. При этом выполняется блок `finally` и вызывается метод `close()`. Данный метод сам может генерировать исключение класса `IOException`! При этом исходное исключение будет потеряно и вместо нее будет возвращен объект класса `IOException`.

Лучше всего совсем не генерировать исключения в операциях, связанных с освобождением ресурсов, например в методах `dispose()`, `close()` и т.п. Именно эти методы разработчики чаще всего помещают в блок `finally`. (К сожалению, разработчики класса `InputStream` с нами не согласны.)

Анализ стека

Трассировка стека — это список вызовов методов для данной точки программы. Вы наверняка неоднократно видели эту информацию, она отображается каждый раз, когда при выполнении Java-программы возникает неконтролируемое исключение.

Трассировка стека предоставляет информацию только относительно выражения, сгенерировавшего исключение; оно не обязательно является источником ошибки.

В версиях, предшествующих JDK 1.4, для получения текстового описания трассировки стека применялся метод `printStackTrace()`. Теперь разработчикам доступен метод `getStackTrace()`; он возвращает массив объектов `StackTraceElement`,

которые можно анализировать в программе. Пример использования этого метода приведен ниже.

```
Throwable t = new Throwable();
StackTraceElement[] frames = t.getStackTrace();
for (StackTraceElement frame : frames)
  // анализ frame
```

Класс `StackTraceElement` содержит методы, позволяющие получить имя файла, номер строки кода, а также имя класса и метода. Форматированную строку, содержащую эту информацию, предоставляет метод `toString()`.

В JDK 5.0 был добавлен статический метод `Thread.getAllStackTraces()`, который позволяет получить трассировку стека для всех потоков. Следующий фрагмент кода демонстрирует его применение:

```
Map<Thread, StackTraceElement[]> map = Thread.getAllStackTraces();
for (Thread t : map.keySet())
{
   StackTraceElement[] frames = map.get(t);
   анализ frames
}
```

Более подробные сведения о потоках и интерфейсе `Map` приведены во втором томе.

В листинге 11.1 показан код программы, которая выводит трассировку стека для рекурсивной функции вычисления факториала. Например, при вычислении `factorial(3)` вы получите следующие данные.

```
factorial(3):
StackTraceTest.factorial(StackTraceTest.java:8)
StackTraceTest.main(StackTraceTest.java:23)
factorial(2):
StackTraceTest.factorial(StackTraceTest.java:8)
StackTraceTest.factorial(StackTraceTest.java:14)
StackTraceTest.main(StackTraceTest.java:23)
factorial(1):
StackTraceTest.factorial(StackTraceTest.java:8)
StackTraceTest.factorial(StackTraceTest.java:14)
StackTraceTest.factorial(StackTraceTest.java:14)
StackTraceTest.main(StackTraceTest.java:23)
return 1
return 2
return 6
```

Листинг 11.1. Содержимое файла `StackTraceTest.java`

```
import java.util.*;

/**
    Программа, отображающая результаты
    трассировки стека для рекурсивного метода.
*/
public class StackTraceTest
{
   /**
      Вычисление факториала.
```

```
      @param n Неотрицательное число.
      @return n! = 1 * 2 * . . . * n
   */
   public static int factorial(int n)
   {
      System.out.println("factorial(" + n + "):");
      Throwable t = new Throwable();
      StackTraceElement[] frames = t.getStackTrace();
      for (StackTraceElement f : frames)
         System.out.println(f);
      int r;
      if (n <= 1) r = 1;
      else r = n * factorial(n - 1);
      System.out.println("return " + r);
      return r;
   }

   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter n: ");
      int n = in.nextInt();
      factorial(n);
   }
}
```

API **java.lang.Throwable 1.0**

- `Throwable(Throwable cause)` **1.4**
- `Throwable(String message, Throwable cause)` **1.4**

  Создает объект класса `Throwable`, соответствующий объекту `cause`, который описывает причины возникновения исключения.

- `Throwable initCause(Throwable cause)` **1.4**

  Указывает причину создания объекта. Если причина создания объекта уже задана, генерирует исключение. Возвращает указатель `this`.

- `Throwable getCause()` **1.4**

  Возвращает объект исключения, который был определен как причина создания данного объекта. Если причина создания объекта не была задана, возвращается значение `null`.

- `StackTraceElement[] getStackTrace()` **1.4**

  Возвращает трассировку стека в момент создания объекта.

API **java.lang.Exception 1.0**

- `Exception(Throwable cause)` **1.4**
- `Exception(String message, Throwable cause)`

  Создает объект класса `Exception`, соответствующий объекту `cause`.

java.lang.RuntimeException 1.0

- `RuntimeException(Throwable cause)` **1.4**
- `RuntimeException(String message, Throwable cause)` **1.4**

Создает объект класса `RuntimeException`, соответствующий объекту `cause`.

java.lang.StackTraceElement 1.4

- `String getFileName()`

  Возвращает имя исходного файла, содержащего точку, в которой выполняется элемент. Если информация недоступна, возвращается значение `null`.

- `int getLineNumber()`

  Возвращает номер строки, содержащей точку, в которой выполняется элемент. Если информация недоступна, возвращается число -1.

- `String getClassName()`

  Возвращает полное имя класса, содержащего точку, в которой выполняется элемент.

- `String getMethodName()`

  Возвращает имя метода, содержащего точку, в которой выполняется элемент. Имя конструктора — `<init>`. Имя статического инициализатора — `<clinit>`. Перегруженные методы, имеющие одинаковые имена, не различаются.

- `boolean isNativeMethod()`

  Если точка выполнения элемента находится внутри платформенно-зависимого метода, возвращает значение `true`.

- `String toString()`

  Возвращает форматированную строку, содержащую имя класса и метода, а также имя файла и номер строки (если эта информация доступна).

Обзор обработки ошибок и исключений в языке Java

В программе, представленной в листинге 11.2, генерируется несколько ошибок и перехватываются различные исключения (рис. 11.2).

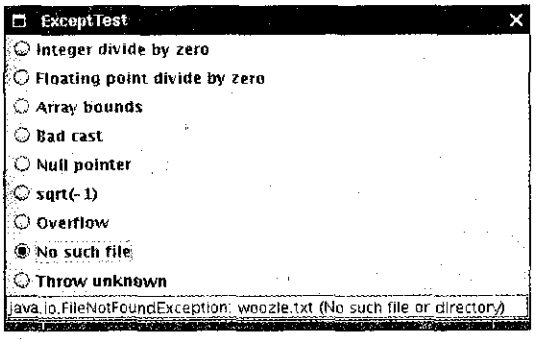


Рис. 11.2. Окно программы, генерирующей исключения

Запустите ее, пощелкайте на переключателях, и вы увидите, как генерируются исключения.

Как известно, ошибки программирования, например выход индекса массива за пределы допустимого диапазона, приводят к генерации исключений `RuntimeException`. Попытка открыть несуществующий файл приводит к возникновению исключения `IOException`. Как ни странно, ошибки, связанные с действиями над числами с плавающей точкой, например деление на 0.0 или извлечение квадратного корня из числа –1, не порождают исключений. (При целочисленном делении на 0 возникает исключение `ArithmeticException`.)

Мы перехватываем исключения, генерируемые методом `actionPerformed()` в методе `fireActionPerformed`, для переключателей и отображаем их в поле редактирования. Однако в заголовке метода `actionPerformed()` не указано, что он генерирует проверяемые исключения. Следовательно, обработчик переключателя No such file должен перехватывать исключение класса `IOException`.

После щелчка на Throw unknown генерируется объект класса `UnknownError`. Этот класс не расширяет класс `Exception`, поэтому наша программа его не перехватывает. Вместо этого код пользовательского интерфейса выводит на консоль сообщение об ошибке и содержимое стека.

Листинг 11.2. Содержимое файла `ExceptTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;

public class ExceptTest
{
   public static void main(String[] args)
   {
      ExceptTestFrame frame = new ExceptTestFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с панелью для тестирования различных исключений.
*/
class ExceptTestFrame extends JFrame
{
   public ExceptTestFrame()
   {
      setTitle("ExceptTest");
      ExceptTestPanel panel = new ExceptTestPanel();
      add(panel);
      pack();
   }
}

/**
    Панель с переключателями для запуска фрагментов кода
    и анализа исключений.
```

```
*/
class ExceptTestPanel extends Box
{
   public ExceptTestPanel()
   {
      super(BoxLayout.Y_AXIS);
      group = new ButtonGroup();

      // Добавление переключателей.

      addRadioButton("Integer divide by zero", new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               a[1] = 1 / (a.length - a.length);
            }
         });

      addRadioButton("Floating point divide by zero", new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               a[1] = a[2] / (a[3] - a[3]);
            }
         });

      addRadioButton("Array bounds", new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               a[1] = a[10];
            }
         });

      addRadioButton("Bad cast", new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               a = (double[])event.getSource();
            }
         });

      addRadioButton("Null pointer", new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               event = null;
               System.out.println(event.getSource());
            }
         });
```

```
      addRadioButton("sqrt(-1)", new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               a[1] = Math.sqrt(-1);
            }
         });

      addRadioButton("Overflow", new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               a[1] = 1000 * 1000 * 1000 * 1000;
               int n = (int) a[1];
            }
         });

      addRadioButton("No such file", new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               try
               {
                  InputStream in = new
                     FileInputStream("woozle.txt");
               }
               catch (IOException e)
               {
                  textField.setText(e.toString());
               }
            }
         });

      addRadioButton("Throw unknown", new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               throw new UnknownError();
            }
         });

      // Добавление поля редактирования для отображения
      // информации об исключениях.
      textField = new JTextField(30);
      add(textField);
   }

   /**
      Помещает на панель переключатель с указанным
      обработчиком.
      @param s Метка переключателя
      @param listener Обработчик
```

```
   */
   private void addRadioButton(String s, ActionListener listener)
   {
      JRadioButton button = new JRadioButton(s, false)
         {
            // Данный метод вызывается для генерации события.
            // Мы переопределяем его для перехвата исключений.
            protected void fireActionPerformed(ActionEvent event)
            {
               try
               {
                  textField.setText("No exception");
                  super.fireActionPerformed(event);
               }
               catch (Exception e)
               {
                  textField.setText(e.toString());
               }
            }
         };

      button.addActionListener(listener);
      add(button);
      group.add(button);
   }

   private ButtonGroup group;
   private JTextField textField;
   private double[] a = new double[10];
}
```

Как использовать исключения

Разные специалисты придерживаются различных мнений относительно использования исключений. Некоторые программисты считают, что контролируемые исключения — это не более чем досадная помеха, другие готовы затратить дополнительное время и усилия на их обработку. На наш взгляд, исключения (особенно контролируемые) — полезный механизм, но применяя его, не стоит слишком увлекаться. В этом разделе мы дадим ряд рекомендаций относительно использования исключений в программах.

1. *Обработка исключений не может заменить собой простую проверку.*

   Для иллюстрации мы написали фрагмент программы, использующей встроенный класс `Stack`. Программа, приведенная в листинге 11.3, делает 1000000 попыток извлечь элемент из пустого стека. Сначала она определяет, пуст ли стек.

   ```
   if (!s.empty()) s.pop();
   ```

   Затем мы заставляем ее извлекать элемент, не обращая внимания на то, что стек пуст. После этого мы перехватываем исключение `EmptyStackException`, которое информирует нас о том, что мы не правы.

   ```
   try()
   ```

```
{
   s.pop();
}
catch (EmptyStackException e)
{
}
```

Время, которое мы затратили на эти операции, указано в табл. 11.1.

Таблица 11.1. Время, затраченное на выполнение операций над стеком

Проверка	Генерация/перехват исключения
110 мс	24550 мс

Как видим, перехват исключения занял намного больше времени, чем простая проверка. Вывод: используйте исключения только в тех случаях, когда это оправдано.

2. *Не увеличивайте без надобности число блоков* `try`.

Многие программисты каждый оператор помещают в отдельный блок `try`.

```
OutputStream out;
Stack s;

for (i = 0; i < 100; i++)
{
   try
   {
      n = s.pop();
   }
   catch (EmptyStackException s)
   {
      // Стек оказался пустым.
   }
   try
   {
      out.writeInt(n);
   }
   catch (IOException e)
   {
      // Проблема при записи в файл.
   }
}
```

При таком подходе размер кода существенно увеличивается. Подумайте о задаче, которую должна решать ваша программа. В данном случае мы хотим извлечь из стека 100 чисел и записать их в файл. (Неважно, зачем нам это надо, — это всего лишь пример.) Сама по себе генерация исключения не решает проблем. Если стек пуст, он не заполнится по мановению волшебной палочки. Если при записи в файл возникает ошибка, она не исчезнет сама собой. Следовательно, имеет смысл поместить в блок `try` *весь фрагмент* программы. Если хотя бы одна из операций даст сбой, можно отказаться от решения всей задачи.

```
try
{
   for (i = 0; i < 100; i++)
   {
      n = s.pop();
      out.writeInt(n);
   }
}
catch (IOException e)
{
   // Проблема при записи в файл.
}
catch (EmptyStackException s)
{
   // Стек оказался пустым.
}
```

Этот фрагмент выполняет одно из своих основных заданий — *отделяет* нормальную работу программы от обработки исключений.

3. *Пользуйтесь возможностями, которые предоставляет иерархия исключений.*

   Не ограничивайтесь генерацией `RuntimeException`. Найдите подходящий подкласс или создайте собственный. Не перехватывайте `Throwable`. При таком подходе ваш код становится трудным для восприятия и поддержки.

   Различайте контролируемые и неконтролируемые исключения. Для обработки контролируемых исключений требуются дополнительные усилия, поэтому не применяйте их для оповещения об ошибках логики. (В библиотеке поддержки отражения это правило не выполняется. Пользователи часто вынуждены создавать код для перехвата тех исключений, которые гарантированно не возникнут.)

   При необходимости смело преобразуйте один тип исключения в другой, более подходящий в данной ситуации. Например, если вы выполняете разбор целого числа, извлекаемого из файла, перехватывайте `NumberFormatException` и генерируйте вместо него подкласс `IOException` или `MySubsystemException`.

4. *Не заглушайте исключения.*

   При написании программы на языке Java существует большой соблазн заглушать исключения. Допустим, вы пишете метод, вызывающий другой метод, который может сгенерировать исключение один раз в сто лет. Компилятор пожалуется на это, поскольку вы не объявили это исключение в списке `throws` вашего метода. Предположим, вам не хочется указывать ее в списке `throws`, так как компилятор начнет выдавать сообщения об ошибке для всех методов, вызывающих ваш метод. В этом случае исключение можно просто заглушить.

```
public Image loadImage(String s)
{
   try
   {
        Код, способный сгенерировать исключение.
   }
   catch (Exception e)
   {} // Действия не выполняются!
}
```

Теперь ваш код будет скомпилирован. Он будет прекрасно работать, но не в исключительной ситуации. Если она возникнет, то будет проигнорирована. Если вы считаете, что исключение достаточно важно, приложите усилия для его обработки.

5. *Генерируя исключения, не бойтесь показаться излишне жестким.*

   Некоторые программисты избегают генерировать исключения, информирующие об ошибках. Например, если методу переданы некорректные параметры, они предпочитают возвращать фиктивное значение. Например, в случае, когда стек пуст, программист может предусмотреть возврат методом `Stack.pop()` значения `null`, вместо того, чтобы сгенерировать исключение. На наш взгляд, лучше исключение `EmptyStackException` в той точке, где возникла ошибка, чем исключение `NullPointerException` впоследствии.

6. *Не бойтесь передавать исключения другим обработчикам.*

   Некоторые программисты считают себя обязанными перехватывать все исключения. Вызывая метод, генерирующий некоторое исключение, например конструктор класса `FileInputStream` или метод `readLine`, они инстинктивно перехватывают исключения, которые могут быть сгенерированы ими. Зачастую оказывается предпочтительнее передать исключение другому обработчику, а не обрабатывать его самому.

```
public void readStuff(String filename)
  throws IOException // Вполне допустимое решение
{
   InputStream in = new FileInputStream(filename);
   . . .
}
```

   Методы более высокого уровня лучше информируют пользователя об ошибке или отказываются выполнять неправильные операторы.

Листинг 11.3. Содержимое файла `ExceptionalTest.java`

```
import java.util.*;

public class ExceptionalTest
{
   public static void main(String[] args)
   {
      int i = 0;
      int ntry = 1000000;
      Stack s = new Stack();
      long s1;
      long s2;

      // ntry раз выполняется проверка, пуст ли стек.
      System.out.println("Testing for empty stack");
      s1 = new Date().getTime();
      for (i = 0; i <= ntry; i++)
         if (!s.empty()) s.pop();
      s2 = new Date().getTime();
      System.out.println((s2 - s1) + " milliseconds");
```

```
      // ntry раз извлекается элемент из пустого стека
      // и генерируется исключение.
      System.out.println("Catching EmptyStackException");
      s1 = new Date().getTime();
      for (i = 0; i <= ntry; i++)
      {
         try
         {
            s.pop();
         }
         catch(EmptyStackException e)
         {
         }
      }
      s2 = new Date().getTime();
      System.out.println((s2 - s1) + " milliseconds");
   }
}
```

Регистрация ошибок

Программисты часто прибегают к вызовам метода `System.out.println` для отладки кода. Разумеется, после устранения ошибки следует убрать промежуточный вывод и поместить его в другое место для выявления новой неисправности. Для упрощения отладки в пакете JDK 1.4 предусмотрены средства регистрации ошибок. Преимущества использования средств регистрации.

- Все регистрационные записи легко заблокировать и разблокировать.
- Заблокированные регистрационные записи занимают немного места и не влияют на эффективность приложения.
- Регистрационные записи можно направить разным обработчикам, вывести на консоль, записать в файл и т.п.
- Регистраторы и обработчики могут фильтровать записи. Фильтры отбрасывают ненужные записи, используя критерии, предложенные программистом.
- Регистрационные записи допускают форматирование. Например, их можно представить в виде простого текста либо в формате XML.
- Приложения могут использовать несколько регистраторов, имеющих иерархические имена, подобно именам пакета, например `com.mycompany.myapp`.
- По умолчанию параметры настройки средств регистрации задаются в конфигурационном файле. Приложение может изменить этот способ задания параметров.

Простая регистрация

Начнем с наиболее простого случая. Система регистрации управляет регистратором `Logger.global`, который используется вместо `System.out`. Вызовем метод `info` для регистрации информационного сообщения.

```
Logger.global.info("File->Open menu item selected");
```

По умолчанию выводится следующая запись:

```
May 10, 2004 10:12:15 PM LoggingImageViewer fileOpen
INFO: File->Open menu item selected
```

(Обратите внимание на то, что время, а также имена класса и метода включаются автоматически.) Однако, если в соответствующем месте (например, в начале метода `main`) включить приведенное ниже выражение, запись будет заблокирована.

```
Logger.global.setLevel(Level.OFF);
```

Расширенные средства регистрации

Теперь, ознакомившись с "регистрацией для "чайников", перейдем к средствам регистрации, применимым при создании промышленных программ. В профессиональном приложении, как правило, все записи не концентрируются в одном глобальном регистраторе. Вместо этого можно определить свои собственные средства регистрации.

Регистратор создается, когда он вызывается по имени в первый раз.

```
Logger myLogger = Logger.getLogger("com.mycompany.myapp");
```

При последующих вызовах регистратора с тем же именем будет возвращаться тот же объект.

Как и имена пакетов, имена регистраторов образуют иерархию. Фактически они являются еще *более* иерархическими, чем имена пакетов. Если между пакетом и его предком нет семантической связи, то регистратор и его дочерние классы обладают общими свойствами. Например, если в регистраторе "`com.mycompany`" задать уровень регистрации, то дочерний регистратор унаследует этот уровень.

Существует семь уровней регистрации.

- `SEVERE`
- `WARNING`
- `INFO`
- `CONFIG`
- `FINE`
- `FINER`
- `FINEST`

По умолчанию используются первые три уровня. Остальные уровни нужно задавать с помощью метода `setLevel`:

```
logger.setLevel(level.FINE);
```

Теперь будут регистрироваться все сообщения, начиная с уровня `FINE` и выше.

Кроме того, можно использовать константу `Level.ALL` для включения регистрации всех уровней и константу `Level.OFF` для отключения регистрации.

Для всех уровней определены методы регистрации, например:

```
logger.warning(message);
logger.fine(message);
```

Помимо этого, можно воспользоваться методом `log` и явно указывать уровень:

```
logger.log(Level.FINE, message);
```

По умолчанию регистрируются все записи, имеющие уровень INFO и выше. Следовательно, для отладочных сообщений, необходимых для диагностики программы, но совершенно не интересующих пользователя, следует использовать уровни CONFIG, FINE, FINER и FINEST.

Если уровень регистрации превышает значение INFO, необходимо изменить конфигурацию обработчика регистрации. По умолчанию обработчик регистрации блокирует сообщения, имеющие уровень ниже, чем INFO.

Запись регистрации, созданная по умолчанию, состоит из имени класса и метода, содержащего вызов регистратора. Однако, если виртуальная машина оптимизирует процесс выполнений, точная информация о вызываемых методах может стать недоступной. Для уточнения вызывающих класса и метода следует применять метод `logp()`.

```
void logp(Level l, String className, String methodName,
   String message)
```

Для трассировки выполнения программы существуют удобные методы:

```
void entering(String className, String methodName)
void entering(String className, String methodName, Object param)
void entering(String className, String methodName,
   Object[] params)
void exiting(String className, String methodName)
void exiting(String className, String methodName, Object result)
```

Ниже приведен пример использования этих методов.

```
int read(String file, String pattern)
{
   entering("com.mycompany.nylib.Reader", "read",
      new Object[] { file, pattern });
   ...
   exiting("com.mycompany.mylib.Reader", "read", count);
   return count;
}
```

Данные вызовы генерируют записи регистрации, имеющие уровень FINER и начинающиеся строками ENTRY и RETURN.

В будущем методы регистрации будут поддерживать переменное число параметров. Тогда вы сможете использовать вызовы наподобие следующего:

```
logger.entering("com.mycompany.mylib.Reader", "read", file, pattern)
```

Обычно регистрация используется для обработки неожиданных исключений. Есть два удобных метода, позволяющих вставить имя исключения в регистрационную запись:

```
void throwing(String className, String methodName, Throwable t)
void log(Level l, String message, Throwable t)
```

Эти методы, как правило, используются следующим образом:

```
if (. . .)
{
```

```
   IOException exception = new IOException(". . .");
   logger.throwing("com.mycompany.mylib.Reader",
      "read", exception);
   throw exception;
}
и
try
{
   . . .
}
catch (IOException e)
{
   Logger.getLogger("com.mycompany.myapp").
      log(Level.WARNING, "Reading image", e);
}
```

Вызов метода `throwing` регистрирует запись, имеющую уровень `FINER`, и сообщение, которое начинается с `THROW`.

Настройка диспетчера регистрации

Свойства системы регистрации можно изменять, редактируя конфигурационный файл. По умолчанию этот файл находится по следующему адресу:

каталог_исполняющей_системы`/lib/logging.properties`

Если вы хотите использовать другой файл, нужно при запуске приложения установить свойство `java.util.logging.config.file`:

```
java -Djava.util.logging.config.file=конфигурационный_файл класс
```

Вызов метода `System.setproperty(java.util.logging.config.file, file)` из метода `main()` ничего не дает, поскольку диспетчер регистрации инициализируется при запуске виртуальной машины, еще до начала выполнения метода `main()`.

Чтобы изменить уровень регистрации, принятый по умолчанию, нужно отредактировать конфигурационный файл, изменив строку

```
.level=INFO
```

В собственных регистраторах уровни регистрации задаются с помощью строк наподобие следующей:

```
com.mycompany.myapp.level=FINE
```

Иначе говоря, нужно добавить к имени регистратора суффикс `.level`.

Как мы увидим в дальнейшем, регистраторы на самом деле не посылают сообщения на консоль — это забота обработчиков. Для обработчиков также определены уровни. Чтобы увидеть на консоли сообщения, имеющие уровень `FINE`, необходимо выполнить следующую установку:

```
java.util.logging.ConsoleHandler.level=FINE
```

Параметры настройки диспетчера регистрации *не* являются системными свойствами. Запуск программы с опциями `-Dcom.mycompany.myapp.level=FINE` никак не отражается на регистраторе.

Файл, содержащий свойства системы регистрации, обрабатывается классом `java.util.logging.LogManager`. Задав имя подкласса в качестве системного свойства `java.util.logging.manager`, можно указать другой диспетчер регистрации. Кроме того, можно оставить стандартный диспетчер регистрации, пропустив инициализационные установки в файле свойств. Детальное описание класса `LogManager` содержится в документации по API.

Поддержка регистрационных сообщений на разных языках

Иногда возникает необходимость обеспечить отображение регистрационных сообщений на разных языках, чтобы они стали понятны пользователям в других странах. (Обеспечение поддержки разных языков приложением называют локализацией.) Интернационализация приложений обсуждается в главе 10 тома 2. При локализации регистрационных сообщений могут возникать некоторые проблемы.

Информация, необходимая для локализации приложения, содержится в *пакетах ресурсов* (resource bundle). Пакет ресурсов представляет собой набор отображений, предназначенных для адаптации приложения для работы в разных странах (например, в США или Германии). Каждая таблица определяет соответствие между именами и объектами. Например, пакет ресурсов может отображать строку `"readingFile"` в строку `"Reading file"` или `"Achtung! Datei wird eingelesen."`

Программа может содержать несколько пакетов ресурсов, например, один для меню, а другой для регистрационных сообщений. Каждый пакет ресурсов имеет имя (например, `"com.mycompany.logmessages"`). Для того чтобы добавить отражение к пакету ресурсов, надо предоставить файл для соответствующего языка. Отражение для английского языка находится в файле `com/mycompany/logmessages_en.properties`, а сообщение на немецком — в файле `com/mycompany/logmessages_de.properties`. (Здесь пары символов `en` и `de` представляют собой коды языков.) Объединение файлов осуществляется с помощью классов вашего приложения, в частности класс `ResourceBundle` автоматически обнаруживает их. Содержимое файлов представляется в текстовом формате и представляет собой записи, подобные приведенным ниже.

```
readingFile=Achtung! Datei wird eingelesen
renamingFile=Datei wird umbenannt
...
```

При вызове регистратора можно указать конкретный пакет ресурсов.

```
Logger logger = Logger.getLogger(loggerName,
   "com.mycompany.logmessages");
```

Затем для создания регистрационного сообщения нужно задать ключ пакета ресурсов (но не фактическую строку).

```
logger.info("readingFile");
```

Часто возникает необходимость включать в локализованные сообщения какие-либо параметры. В таком случае сообщение должно иметь заполнители `{0}`, `{1}` и т.д. Например, может потребоваться включить в регистрационное сообщение имя файла. Тогда заполнитель используется следующим образом:

```
Reading file {0};
Achtung! Datei {0} wird eingelesen.
```

Заполнители заменяются значениями при вызове одного из следующих методов:

```
logger.log(Level.INFO, "readingFile", fileName);
logger.log(Level.INFO, "renamingFile",
   new Object[]{ oldName, newName });
```

Обработчики

По умолчанию регистраторы посылают сообщения методу `ConsoleHandler`, который выводит их в поток `System.err`. Точнее говоря, регистраторы посылают запись предку обработчика, а его последний наследник (с именем "") содержит метод `ConsoleHandler`.

Как и регистраторы, обработчики имеют уровни. Уровень регистрации записи, подлежащей выводу, должен превышать пороговое значение *и регистратора, и обработчика.* Уровень консольного обработчика по умолчанию задается в файле, содержащем параметры настройки диспетчера регистрации:

```
java.util.logging.ConsoleHandler.level=INFO
```

Для регистрации записи, имеющей уровень `FINE`, необходимо изменить в конфигурационном файле как уровень регистрации, так и уровень обработчика, предусмотренные по умолчанию. Кроме того, можно совсем проигнорировать файл конфигурации и установить свой собственный обработчик.

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
logger.setLevel(Level.FINE);
logger.setUseParentHandlers(false);
Handler handler = new ConsoleHandler();
handler.setLevel(Level.FINE);
logger.addHandler(handler);
```

По умолчанию регистратор посылает сообщения своему обработчику и его предку. Наш регистратор является наследником первоначального регистратора (имеющего имя ""), посылающего на консоль все записи уровня `INFO` и выше. Однако никому не нужно, чтобы все регистрационные записи выводились на экран дважды, поэтому свойство `useParentHandlers` следует задать равным `false`.

Для того чтобы послать регистрационную запись еще куда-нибудь, нужно добавить другой обработчик. С этой целью система регистрации предоставляет в наше распоряжение два класса: `FileHandler` и `SocketHandler`. Обработчик `SocketHandler` передает сообщения на указанные узел по заданному порту. Наибольший интерес для нас представляет обработчик `FileHandler`, помещающий записи в файл.

Записи можно просто передать обработчику `FileHandler` по умолчанию.

```
FileHandler handler = new FileHandler();
logger.addHandler(handler);
```

Записи помещаются в файл `javan.log`, который находится в рабочем каталоге пользователя. Число *n* обеспечивает уникальность этого файла. Если на платформе понятие рабочего каталога пользователя не используется (например, в Windows 95), файл записывается в каталог, предусмотренный по умолчанию, например в каталог `C:\Windows`. Записи хранятся в формате XML. Обычная регистрационная запись имеет такой вид:

```
<record>
  <date>2002-02-04T07:45:15</date>
```

```
  <millis>1012837515710</millis>
  <sequence>1</sequence>
  <logger>com.mycompany.myapp</logger>
  <level>INFO</level>
  <class>com.mycompany.mylib.Reader</class>
  <method>read</method>
  <thread>10</thread>
  <message>Reading file corejava.gif</message>
</record>
```

Поведение обработчика `FileHandler`, определенное по умолчанию, можно изменить, задав другие параметры настройки диспетчера регистрации (табл. 11.2) или используя другой конструктор.

Программисты, как правило, не используют имя файла регистрации, предлагаемое по умолчанию. Следовательно, нужно задать другой шаблон, например `%h/myapp.log` (переменные шаблона описаны в табл. 11.3).

Если несколько приложений (или несколько копий одного и того же приложения) используют один и тот же файл регистрации, следует установить флаг, управляющий добавлением данных в конец файла. Кроме того, можно использовать шаблон имени файла `%u` так, чтобы каждое приложение создавало уникальную копию файла регистрации.

Хорошей идеей является также "ротация" имени файла. В этом случае имена файлов регистрации формируются так: `myapp.log.0`, `myapp.log.1`, `myapp.log.2` и т.д. Как только размер файла превысит допустимый предел, наиболее старый файл регистрации удаляется, остальные переименовываются, а новый файл получает имя с номером 0.

Таблица 11.2. Параметры настройки обработчика `FileHandler`

Свойство	Описание	Значение по умолчанию
`java.util.logging.FileHandler.level`	Уровень обработчика	`Level.ALL`
`java.util.logging.FileHandler.append`	Определяет, должен ли обработчик добавлять записи в существующий файл или открывать новый файл при каждом новом запуске программы	`false`
`java.util.logging.FileHandler.limit`	Приблизительная оценка максимального объема файла. При превышении этого размера открывается новый файл (0 — объем не ограничен)	В классе `FileHandler` — 0 (неограниченное количество), в конфигурации диспетчера регистрации по умолчанию — 50000
`java.util.logging.FileHandler.pattern`	Шаблон имени файла регистрации (см. табл. 11.3)	`%h/java%u.log`
`java.util.logging.FileHandler.count`	Количество файлов регистрации, используемых для ротации	1 (ротация не выполняется)
`java.util.logging.FileHandler.filter`	Класс фильтра	Фильтрация отсутствует
`java.util.logging.FileHandler.encoding`	Кодировка	Кодировка, принятая в текущей операционной системе
`java.util.logging.FileHandler.formatter`	Средство форматирования записей	`java.util.logging.XMLformatter`

Многие программисты используют средства регистрации для технической поддержки. Если в какой-то момент программа начинает работать некорректно, пользователь может обратиться к файлу регистрации. Чтобы это стало возможным, надо установить флаг, управляющий добавлением данных к имеющемуся файлу, использовать ротацию либо применить оба подхода одновременно.

Таблица 11.3. Переменные шаблона в имени файла регистрации

Переменная	Описание
`%h`	Значение системного свойства `user.home`
`%t`	Временный системный каталог
`%u`	Уникальный номер, позволяющий избежать конфликта имен
`%g`	Генерация номера для ротации
`%%`	Символ `%`

Можно также определять свои собственные обработчики, расширяя класс `Handler` или `StreamHandler`. Такой обработчик будет определен в программе, приведенной в качестве примера. Этот обработчик отображает записи на экране (рис. 11.3).

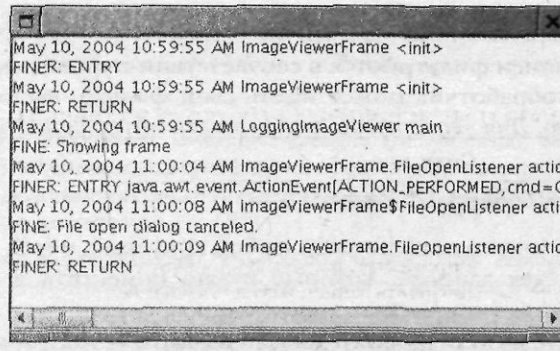


Рис. 11.3. Обработчик, отображающий записи на экране

Обработчик расширяет класс `StreanHandler` и устанавливает поток, содержащий метод `write()`, который выводит информацию в текстовую область.

```
class WindowHandler extends StreamHandler
{
   public WindowHandler()
   {
      . . .
      final JTextArea output = new JTextArea();
      setOutputStream(new
         OutputStream()
         {
            public void write(int b) {} // Не вызывается.
            public void write(byte[] b, int off, int len)
            {
               output.append(new String(b, off, len));
            }
```

```
        });
    }
    . . .
}
```

При таком подходе возникает лишь одна проблема — обработчик помещает записи в буфер и выводит их в поток только тогда, когда буфер переполняется. Следовательно, нужно переопределить метод `publish()`, чтобы выводить содержимое буфер после каждой записи.

```
class WindowHandler extends StreamHandler
{
   ...
   public void publish(LoigRecord record)
   {
      super.publish(record);
      flush();
   }
}
```

При желании можно написать более "экзотические" обработчики. Для этого достаточно расширить класс `Handler` и определить методы `publish()`, `flush()` и `close()`.

Фильтры

По умолчанию записи фильтруются в соответствии с уровнями регистрации. Каждый регистратор и обработчик может иметь свой фильтр, выполняющий дополнительную фильтрацию. Для этого нужно реализовать интерфейс `Filter` и определить следующий метод:

```
boolean isloggable(LogRecord record)
```

Для анализа записей можно использовать любой критерий; метод должен возвращать значение `true` для тех записей, которые нужно поместить в файл регистрации. Например, фильтр может пропускать только записи, созданные в начале выполнения метода и при возврате из него. Фильтр должен вызвать метод `record.getMessage()` и проверить, начинается запись с *ENTRY* или *RETURN*.

Для того чтобы задать фильтр в регистраторе или обработчике, вызовите метод `setFilter()`. Обратите внимание на то, что в каждый момент времени в вашем распоряжении может быть только один фильтр.

Средства форматирования

Классы `ConsoleHandler` и `FileHandler` порождают записи регистрации в текстовом и XML-форматах. Однако существует возможность определить свой собственный формат. Для этого необходимо расширить класс `Formatter` и переопределить следующий метод:

```
String format(logRecord record)
```

Затем следует форматировать запись любым выбранным способом и вернуть получившуюся строку. В методе `format()` может потребоваться вызов метода

```
String formatMessage(LogRecord record)
```

Этот метод форматирует сообщение, содержащееся в регистрационной записи, заменяя параметры и выполняя локализацию.

Многие форматы файлов (например, XML) предполагают существование "головы" и "хвоста" файла, охватывающих форматированные записи. В этом случае следует переопределить методы

```
String getHead(Handler h)
String getTail(Handler h)
```

В заключение следует вызвать метод `setFormatter()`, инсталлирующий средства форматирования для обработчика.

Имея столько возможностей для регистрации, легко запутаться. Ниже приведены краткие рекомендации по выполнению основных операций.

Способы регистрации

1. Для простых приложений выбирайте один регистратор. Желательно, чтобы имя регистратора совпадало с именем основного пакета приложения, например `com.mycompany.myprog`. Создать регистратор можно с помощью вызова

   ```
   Logger.getLogger("com.mycompany.myprog");
   ```

   Для удобства можно добавить в классы, интенсивно использующие регистрацию, статические поля

   ```
   private static final Logger logger = Logger.getLogger("com.mycompany.myprog");
   ```

2. По умолчанию все сообщения, имеющие уровень `INFO` и выше, выводятся на консоль. Пользователи могут изменить конфигурацию, предусмотренную по умолчанию, но, как мы уже видели, это довольно сложный процесс. Следовательно, лучше задать более оправданные установки приложения.

   Приведенный ниже код гарантирует, что все сообщения будут зарегистрированы в файле, связанном с конкретным приложением. Поместите его в метод `main()` вашего приложения.

   ```
   if (System.getProperty("java.util.loging.config.class")== null
     && System.getProperty("java.util.loging.config.file")== null)
   {
      try
      {
         Logger.getLogger("").setLevel(Level.ALL);
         final int LOG_ROTATION_COUNT = 10;
         Handler handler = new FileHandler(%h/myapp.log,
            0, LOG_ROTATION_COUNT);
         Logger.getLogger("").addHandler(handler);
      }
      catch (IOException e)
      {
         logger.log (Level.SEVERE, "Can' create log file handler", e);
      }
   }
   ```

3. Теперь все готово для регистрации сообщений. Помните, что все сообщения, имеющие уровень `INFO`, `WARNING` и `SEVERE`, выводятся на консоль. Следовательно, эти уровни нужно зарезервировать для сообщений, имеющих ценность для пользователей вашей программы. Уровень `FINE` лучше выделить для сообщений, предназначенных программистам. Там, где вы обычно использовали метод `System.out.println()`, регистрируйте сообщения:

```
logger.fine("File open dialog canceled");
```

Следует регистрировать также неожиданные исключения, например, так:

```
try
{
   ...
}
catch (SomeException exception)
{
   Logger.getlogger('com.mycompany.myprog".log(
      Level.FINE, "explanation", exception);
}
```

В листинге 11.4 приведен код программы, предназначенной для просмотра изображений, в которой предусмотрена регистрация событий.

Листинг 11.4. Содержимое файла `LoggingImageWindow.java`

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.io.*;
import java.util.logging.*;
import javax.swing.*;

/**
   Модификация программы для просмотра изображений.
   В программе предусмотрена регистрация событий.
*/
public class LoggingImageViewer
{
   public static void main(String[] args)
   {
      if (System.getProperty
            ("java.util.logging.config.class") == null
         && System.getProperty("java.util.logging.config.file") == null)
      {
         try
         {
            Logger.getLogger("").setLevel(Level.ALL);
            final int LOG_ROTATION_COUNT = 10;
            Handler handler = new
               FileHandler("%h/LoggingImageViewer.log",
               0, LOG_ROTATION_COUNT);
            Logger.getLogger("").addHandler(handler);
         }
         catch (IOException e)
         {
            Logger.getLogger("com.horstmann.corejava").
               log(Level.SEVERE,
               "Can't create log file handler", e);
         }
      }

      Handler windowHandler = new WindowHandler();
```

```
      windowHandler.setLevel(Level.ALL);
      Logger.getLogger("com.horstmann.corejava").
         addHandler(windowHandler);

      JFrame frame = new ImageViewerFrame();
      frame.setTitle("LoggingImageViewer");
      frame.setSize(300, 400);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      Logger.getLogger("com.horstmann.corejava").
         fine("Showing frame");
      frame.setVisible(true);
   }
}

/**
   Фрейм для вывода изображений.
*/
class ImageViewerFrame extends JFrame
{
   public ImageViewerFrame()
   {
      logger.entering("ImageViewerFrame", "<init>");
      // Создание строки меню.
      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);

      JMenu menu = new JMenu("File");
      menuBar.add(menu);

      JMenuItem openItem = new JMenuItem("Open");
      menu.add(openItem);
      openItem.addActionListener(new FileOpenListener());

      JMenuItem exitItem = new JMenuItem("Exit");
      menu.add(exitItem);
      exitItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               logger.fine("Exiting.");
               System.exit(0);
            }
         });

      // Для вывода изображений используется метка.
      label = new JLabel();
      add(label);
      logger.exiting("ImageViewerFrame", "<init>");
   }

   private class FileOpenListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
```

```
            logger.entering("ImageViewerFrame.FileOpenListener",
                "actionPerformed", event);

            // Создание окна для выбора файлов.
            JFileChooser chooser = new JFileChooser();
            chooser.setCurrentDirectory(new File("."));

            // Допустимыми считаются файлы с расширением .gif
            chooser.setFileFilter(new
               javax.swing.filechooser.FileFilter()
               {
                  public boolean accept(File f)
                  {
                     return f.getName().toLowerCase().
                              endsWith(".gif") || f.isDirectory();
                  }
                  public String getDescription()
                  {
                     return "GIF Images";
                  }
               });

            // Отображение диалогового окна для выбора файлов.
            int r = chooser.showOpenDialog(ImageViewerFrame.this);

            // Если файл выбран, устанавливается пиктограмма
            if (r == JFileChooser.APPROVE_OPTION)
            {
               String name = chooser.getSelectedFile().getPath();
               logger.log(Level.FINE, "Reading file {0}", name);
               label.setIcon(new ImageIcon(name));
            }
            else
               logger.fine("File open dialog canceled.");
            logger.exiting("ImageViewerFrame.FileOpenListener",
               "actionPerformed");
         }
      }

      private JLabel label;
      private static Logger logger =
         Logger.getLogger("com.horstmann.corejava");
}

/**
   Обработчик для вывода в окне регистрационных записей.
*/
class WindowHandler extends StreamHandler
{
   public WindowHandler()
   {
      frame = new JFrame();
      final JTextArea output = new JTextArea();
      output.setEditable(false);
      frame.setSize(200, 200);
      frame.add(new JScrollPane(output));
```

```
      frame.setFocusableWindowState(false);
      frame.setVisible(true);
      setOutputStream(new
         OutputStream()
         {
            public void write(int b) {} // Не вызывается.
            public void write(byte[] b, int off, int len)
            {
               output.append(new String(b, off, len));
            }
         });
   }

   public void publish(LogRecord record)
   {
      if (!frame.isVisible()) return;
      super.publish(record);
      flush();
   }

   private JFrame frame;
}
```

API **java.util.logging.Logger 1.4**

- `Logger getLogger(String loggerName)`
- `Logger getLogger(String loggerName, String bundleName)`

  Возвращает регистратор с указанным именем. Если регистратор не существовал, он создается.

  Параметры `loggerName` Иерархическое имя регистратора, например `com.mycompany.myapp`.

  `bundleName` Имя пакета ресурсов для просмотра локализованных сообщений

- `void severe(String message)`
- `void warning(String message)`
- `void info(String message)`
- `void config(String message)`
- `void fine(String message)`
- `void finer(String message)`
- `void finest(String message)`

  Регистрирует запись, имеющую уровень, соответствующий имени метода, и содержащую заданное сообщение.

- `void entering(String className, String methodName)`
- `void entering(String className, String methodName, Object param)`
- `void entering(String className, String methodName, Object[] param)`
- `void exiting(String className, String methodName)`
- `void exiting(String className, String methodName, Object result)`

Регистрирует запись, описывающую вход и выход из метода с заданными параметрами или возвращаемым значением.

- `void throwing(String className, String methodName, Throwable t)`

Регистрирует запись, описывающую процесс генерации объекта исключения.

- `void log(Level level, String message)`
- `void log(Level level, String message, Object obj)`
- `void log(Level level, String message, Object[] obj)`
- `void log(Level level, String message, Throwable t)`

Регистрирует запись, имеющую заданный уровень и содержащую указанное сообщение. Запись может содержать обычные объекты или объект исключения. Для этой цели в сообщении предусмотрены заполнители `{0}`, `{1}` и т.д.

- `void logp(Level level, String className, String methodName, String message)`
- `void logp(Level level, String className, String methodName, String message, Object obj)`
- `void logp(Level level, String className, String methodName, String message, Object[] objs)`
- `void logp(Level level, String className, String methodName, String message, Throwable t)`

Регистрирует запись, имеющую заданный уровень и содержащую и информацию о вызове, и указанное сообщение. Запись может включать обычные объекты или объекты исключений.

- *`void logrb(Level level, String className, String methodName,`* `String bundleName, String message)`
- `void logrb(Level level, String className, String methodName, String bundleName, String message, Object obj)`
- `void logrb(Level level, String className, String methodName, String bundleName, String message, Object[] objs)`
- `void logrb(Level level, String className, String methodName, String bundleName, String message, Throwable t)`

Регистрирует запись, имеющую заданный уровень, содержащую информацию о вызове и указанное сообщение. Запись может включать обычные объекты или объекты исключений.

- `Level getLevel()`
- `void setLevel(Level l)`

Возвращает и устанавливает уровень регистратора.

- `Logger getParent()`
- `void setParent(Logger l)`

Возвращает и устанавливает родительский регистратор.

- `Handler[] getHandlers()`

Возвращает все обработчики для регистратора.

- `void addHandler(Handler h)`
- `void removeHandler(Hadnler h)`

Добавляет и удаляет обработчик для регистратора.

- `boolean getUseParentHandlers()`
- `void setUseParentHandlers(boolean b)`

Возвращает и устанавливает свойство, управляющее использованием родительского обработчика. Если значение этого свойства равно `true`, регистратор направляет все зарегистрированные записи обработчикам своего родительского регистратора.

- `Filter getFilter()`
- `void setFilter(Filter f)`

Возвращает и устанавливает фильтр для регистратора.

java.util.logging.Handler 1.4

- `abstract void publish(LogRecord record)`

Посылает сообщение по указанному адресу.

- `abstract void flush()`

Выводит данные из буфера.

- `abstract void close()`

Выводит данные из буфера и освобождает все ресурсы.

- `Filter getFilter()`
- `void setFilter(Filter f)`

Возвращает и устанавливает фильтр для обработчика.

- `Formatter getFormatter()`
- `void setFormatter(Formatter f)`

Возвращает и устанавливает средства форматирования для обработчика.

- `Level getlevel()`
- `void setLevel(Level l)`

Возвращает и устанавливает уровень обработчика.

java.util.logging.ConsoleHandler 1.4

- `ConsoleHandler()`

Создает новый консольный обработчик.

java.util.logging.FileHandler 1.4

- `FileHandler(String pattern)`
- `FileHandler(String pattern, boolean append)`
- `FileHandler(String pattern, int limit, int count)`

- `FileHandler(String pattern, int limit, int count, boolean append)`

  Создают файловый обработчик.

Параметры		
	`pattern`	Шаблон для создания имени файла регистрации (см. табл. 11.3)
	`limit`	Приблизительная оценка максимального объема файла регистрации. При превышении этого объема открывается новый файл
	`count`	Количество файлов при выполнении ротации
	`append`	Значение `true`, если вновь созданный объект файлового обработчика должен выводить данные в существующий файл регистрации

`java.util.logging.LogRecord 1.4`

- `Level getLevel()`

  Возвращает уровень регистрации для записи.

- `String getLoggerName()`

  Возвращает имя регистратора, который создал запись.

- `ResourceBundle getResourceBundle()`
- `String getResourceBundleName()`

  Возвращают пакет ресурсов, предназначенный для локализации сообщения, или его имя. Если пакет не предусмотрен, возвращается значение `null`.

- `String getMessage()`

  Возвращает "низкоуровневое" сообщение, не прошедшее локализацию и форматирование.

- `Object[] getParameters()`

  Возвращает параметры или значение `null` при их отсутствии.

- `Throwable getThrown()`

  Возвращает объект сгенерированного исключения или `null`, если объект не существует.

- `String getSourceClassName()`
- `String getSourceMethodName()`

  Определяют местоположение кода, зарегистрировавшего данную запись. Эта информация может быть предоставлена кодом регистрации или автоматически выведена на основании анализа стека выполнения программы. Если код регистрации содержит неверное значение или программа была оптимизирована так, что определить местоположение кода невозможно, информация может оказаться неточной.

- `long getMillis()`

  Определяет время создания записи в миллисекундах, отсчитывая его от 1970 года.

- `long getSequenceNumber()`

  Возвращает уникальный номер записи.

- `int getThreadID()`

  Возвращает уникальный идентификационный номер потока, в котором была создана запись. Эти идентификационные номера присваиваются классом `LogRecord` и не имеют отношения к другим идентификаторам потоков.

java.util.logging.Filter 1.4

- `boolean isLoggable(LogRecord record)`

  Если указанная запись должна быть зарегистрирована, возвращает значение `true`.

java.util.logging.Formatter 1.4

- `abstract String format(LogRecord record)`

  Возвращает строку, полученную в результате форматирования регистрационной записи.

- `String getHead(Handler h)`
- `String getTail(Handler h)`

  Возвращают строки, которые должны появиться в начале и в конце документа, содержащего регистрационную запись. Эти методы определены в суперклассе `Formatter` и по умолчанию возвращают пустую строку. При необходимости их следует переопределить.

- `String format(LogRecord record)`

  Возвращает локализованное и форматированное сообщение, содержащееся в регистрационной записи.

Диагностические утверждения

Диагностические утверждения (assertion) — это широко распространенные идиомы, применяемые для обеспечения надежности программирования. Допустим, вы убеждены в том, что конкретное свойство уже задано, и ссылаетесь на него в своей программе. Предположим, что вы вычисляете следующее выражение:

```
double y = Math.sqrt(x);
```

Вы уверены, что число `x` — неотрицательное. Возможно, оно получено в результате других вычислений, которые не могут порождать отрицательные числа, либо является параметром метода, которому можно передавать лишь положительные числа. Но вы все же хотите еще раз проверить свои предположения, чтобы в ходе выполнения программы не возникло ошибки. Разумеется, можно было бы просто сгенерировать исключение.

```
if (x < 0) throw new IllegalArgumentException("x < 0 ");
```

Однако у такого подхода есть досадный недостаток — код остается в программе, даже если тестирование давно закончено. Если в тексте программы расставлено много таких проверок, ее выполнение может замедлиться.

Механизм диагностических утверждений позволяет вставлять в текст программы проверки, предназначенные для ее тестирования, а затем удалять их из окончательного варианта.

В пакете JDK 1.4 в языке Java появилось новое ключевое слово — `assert`. Оно имеет две формы:

```
assert условие;
```
и
```
assert условие : выражение;
```

Оба оператора вычисляют некое условие и генерируют исключение `AssertionError`, если оно не выполняется. Во втором операторе выражение передается конструктору объекта класса `AssertionError` и превращается в строку сообщения.

Единственная цель выражения при указании ключевого слова `assert` — создать строку сообщения. Объект класса `AssertionError` не хранит фактическое значение выражения, поэтому его невозможно использовать в дальнейшем. В документации, поставляемой совместно с пакетом JDK 1.4, непринужденно говорится, что это сделано для того, чтобы "утешить программистов, если диагностическое утверждение окажется неверным".

Чтобы проверить, является ли значение `x` неотрицательным числом, можно использовать оператор

```
assert x >= 0;
```

Можно также передать фактическое значение переменной *x* объекту `AssertionError`, *чтобы он впоследствии вывел его на экран.*

```
assert x >= 0 : x;
```

В языке C макрос `assert` превращает проверяемое условие в строку, которая выводится на экран, если оно не выполняется. Например, если выражение `assert(x>=0)` имеет значение `false`, на экран выводится строка, сообщающая, что условие `x >= 0` не выполняется. В языке Java условие не является неотъемлемой частью сообщения об ошибке. Если вы хотите увидеть его, передайте его в виде строки объекту класса `AssertionError`: `assert x >= 0 : "x >= 0".`

Поскольку ключевое слово `assert` появилось совсем недавно, необходимо специально указать на него компилятору. Для этого надо задать опцию `-source 1.4`, например:

```
javac -source 1.4 MyClass.class
```

Начиная с версии JDK 5.0, поддержка диагностических утверждений по умолчанию включена.

Подключение и отключение диагностических утверждений

По умолчанию диагностические утверждения считаются заблокированными. Разблокировать их в ходе выполнения программы можно с помощью опции `-enableassertions` или `-ea`.

```
java -enableassertions MyApp
```

Обратите внимание на то, что для подключения и отключения диагностических утверждений не нужно заново компилировать программу. Это задача *загрузчика классов*. Если диагностические утверждения заблокированы, загрузчик классов игнорирует их код, чтобы не замедлять выполнение программы.

Подключать диагностические утверждения можно в отдельных классах или в целых пакетах.

```
java -ea:MyClass -ea:com.mycompany.mylib ... MyApp
```

Эта команда подключает диагностические утверждения к классу `MyClass`, а также ко всем классам пакета `com.mycompany.mylib` и *его подпакетов*. Опция `-ea...` подключает диагностические утверждения ко всем классам пакета по умолчанию.

Кроме того, можно отключить диагностические утверждения в определенных классах и пакетах. Для этого достаточно задать опцию `-diableassertions` или `-da`.

```
java -ea:... -da:MyClass MyApp
```

Некоторые классы загружаются не загрузчиком классов, а непосредственно виртуальной машиной. Для выборочного включения или отключения диагностических утверждений, содержащихся в этих классах, также можно использовать опции `-ea` и `-da`. Однако эти опции, включающие и отключающие все диагностические утверждения одновременно, не могут применяться к "системным классам" без загрузчика. Для включения диагностических утверждений, находящихся в системных классах, следует использовать опцию `-enablesystemassertions/-esa`.

Можно даже управлять состоянием диагностических утверждений с помощью программных средств. Подробности вы найдете в документации по API.

Рекомендации по использованию диагностических утверждений

В языке Java есть три механизма обработки системных сбоев.

- Генерация исключений.
- Регистрация ошибок.
- Использование диагностических утверждений.

Когда следует применять диагностические утверждения? Чтобы правильно ответить на этот вопрос, учтите следующее.

- Диагностические утверждения оказываются ложными, если произошла неустранимая ошибка.
- Диагностические утверждения включаются только на период отладки и тестирования программы. (Иногда это шутливо формулируют так: "одевайте спасательный жилет, плавая у берега, — в открытом море он вам не понадобится".)

Итак, диагностические утверждения не следует применять в качестве индикаторов несущественных и легко исправимых ошибок. Они предназначены для локализации серьезных неполадок во время тестирования.

Представим себе следующий сценарий: проверку параметров метода. Следует ли применять диагностические утверждения для проверки выхода индекса за пределы допустимого диапазона или для распознавания нулевой ссылки? Чтобы ответить на этот вопрос, необходимо обратиться к документации, описывающей метод. Например, рассмотрим метод `Array.sort` из стандартной библиотеки.

```
/**
   Упорядочивает указанный отрезок заданного массива
   в порядке возрастания. Упорядочиваемый отрезок начинается
   с индекса fromIndex и заканчивается индексом, предшествующим
   toIndex, т.е. элемент с индексом toIndex
   упорядочению не подлежит
   @param упорядочиваемый массив
   @param fromIndex индекс первого элемента (включительно)
   упорядочиваемого отрезка
   @param toIndex индекс первого из элементов, лежащего
   за упорядочиваемым отрезком
   @throws IllegalArgumentException, если fromindex > toIndex
   @throws ArrayIndexOutOfBoundsException, если fromIndex < 0
     или toIndex > a.length
*/
   static void sort[] a, int fromIndex, int toIndex
```

В документации утверждается, что этот метод генерирует исключение, если значение индекса задано некорректно. Это поведение метода является частью соглашения, который он заключает с вызывающим методом. Если вы реализуете этот метод, то должны придерживаться соглашения и генерировать соответствующее исключение. В этом случае диагностические утверждения не подходят.

Следует ли применять их для распознавания нулевой ссылки? Как и прежде, ответ отрицательный. В документации о методе ничего не сказано о его реакции, если параметр a равен константе `null`. Вызывающий модуль имеет основания предполагать, что в этом случае метод будет выполнен и не будет генерировать исключение `AssertionError`.

Допустим, однако, что соглашения слегка изменены.

```
@param a упорядочиваемый массив. (Не должен равняться null.)
```

Теперь вызывающий метод знает, что передавать нулевую ссылку данному методу запрещено. Следовательно, метод может начинаться с диагностического утверждения

```
assert(a != null);
```

Специалисты называют подобные соглашения *предусловиями* (precondition). В начале метод не накладывал на массив никаких предварительных условий, так как предполагалось, что он будет корректно работать во всех случаях. Модифицированный метод имеет одно предусловие – параметр a не должен принимать значение `null`. Если вызывающий модуль нарушает указанное предусловие, то соглашение оказывается нарушенным и метод отказывается выполнять работу. Фактически имея диагностическое утверждение, при неправильном вызове метод может повести себя непредсказуемо. Иногда он генерирует исключение `assertionError`, а иногда – `NullPointerexception`, в зависимости от конфигурации загрузчика классов.

В JDK 5.0 метод `Array.sort`, получив значение `null`, генерирует исключение `NullPointerException`. Это можно считать либо ошибкой документации, либо ошибкой реализации.

Многие программисты оформляют свои предположения о работе программы в виде комментариев. Так, в документации по JDK можно найти такой пример:

```
if (i % 3 == 0)
   ...
else if (i % 3 == 1)
   ...
else // (i % 3 == 2)
   ...
```

В данном случае намного разумнее использовать диагностические утверждения.

```
if (i % 3 == 0)
   ...
else if (i % 3 == 1)
   ...
else
{
   assert(i % 3 == 2)
   ...
```

Разумеется, было бы еще более разумно тщательнее сформулировать проверки. Какие значения может принимать выражение `i % 3`? Если переменная `i` положительна, остаток может равняться 0, 1 или 2. Если отрицательна — остаток принимает значения –1 или –2. Таким образом, намного логичнее предположить просто, что переменная `i` не является отрицательным числом. Логично использовать диагностическое утверждение следующего вида:

```
assert(i >= 0);
```

Его следует разместить перед оператором `if`.

Итак, диагностические утверждения — это тактическое средство, предназначенное для тестирования и отладки, а регистрация — стратегический механизм, сопровождающий программу на протяжении всего ее существования.

java.lang.ClassLoader 1.0

- `void setDefaultAssertionStatus(boolean b)` **1.4**

  Включает и отключает диагностические утверждения во всех классах, загруженных указанным загрузчиком классов, в которых статус диагностических утверждений не задан явно.

- `void setClassAssertionStatus(String className, boolean b)` **1.4**

  Включает и отключает диагностические утверждения в заданном классе и его подклассах.

- `void setPackageAssertionStatus(String packageName, boolean b)` **1.4**

  Включает и отключает диагностические утверждения во всех классах заданного пакета и его подпакетов.

- `void clearAssertionStatus()` **1.4**

  Удаляет все диагностические утверждения, статус которых задан явно, и блокирует все диагностические утверждения в классах, загруженных данным загрузчиком.

Приемы отладки

Допустим, что вы написали программу и оснастили ее средствами для перехвата и соответствующей обработки исключений. Затем вы запускаете ее на выполнение, и она работает неправильно. Что теперь? (Если перед вами никогда не возникала такая проблема, можете не читать остальную часть главы.)

Разумеется, лучше всего было бы иметь под рукой удобный и мощный отладчик. Такие отладчики являются частью профессиональных сред программирования, например Eclipse, NetBeans или JBuilder. Однако при работе с новой версией языка Java, который еще не поддерживается этими интегрированными средами программирования, а также при работе на непривычной платформе приходится уделять отладке очень много времени, расставляя по всей программе выражения для вывода или регистрации.

Рекомендации по отладке программ

Если вам самому придется заняться отладкой, воспользуйтесь нашими советами.

1. Значение любой переменной можно вывести с помощью приведенного ниже выражения.

   ```
   System.out.println("x = " + x);
   ```

   Поместить это значение в регистрационный файл можно следующим образом:

   ```
   Logger.global.info("x=" + x);
   ```

   Если в переменной x содержится числовое значение, оно преобразуется в строку. Если переменная x — объект, то вызывается метод `toString`. Состояние объекта, используемого в качестве неявного параметра, поможет определить следующее выражение:

   ```
   Logger.global.info("this=" + this);
   ```

   В большинстве классов языка Java метод `toString` переопределяется таким образом, чтобы предоставить пользователю как можно больше полезной информации. Это очень удобно для отладки. В своих классах следует самому позаботиться об этом.

2. Один малоизвестный, но очень полезный прием заключается в том, что в каждый класс можно включить отдельный метод `main()` и разместить в нем код, позволяющий протестировать этот класс отдельно от других.

   ```
   public class MyClass
   {
      методы и поля
      ...
      public static void main(String[] args)
      {
         код для проверки
      }
   }
   ```

   Создайте несколько объектов, вызовите все методы и проверьте, правильно ли они работают. Это делается с помощью методов `main()`; интерпретатор вызывается для каждого файла по отдельности. Если выполняется аплет, ни один из этих методов `main()` никогда не будет вызван, если выполняется приложение, то интерпретатор вызовет лишь метод `main()` из стартового класса.

3. Если предыдущая рекомендация пришлась вам по душе, попробуйте поработать с продуктом JUnit, который можно скопировать, обратившись по адресу `http://junit.org`. JUnit представляет собой популярную систему модульного тестирования, которая существенно упрощает работу по созданию наборов сценариев тестирования. Запускайте тесты после каждой модификации класса. При обнаружении неисправности добавляйте новый сценарий тестирования.
4. Proxy-объект регистрации представляет собой экземпляр подкласса, который перехватывает вызовы методов, создает регистрационные данные и обращается к суперклассу. Например, если у вас возникли проблемы с методом `setBackground()`, устанавливающем цвет фона панели, вы можете создать proxy-объект в виде безымянного класса.

```
JPanel panel = new
   JPanel()
   {
      public void setBackground(Color c)
      {
         Logger.global.info("setBackground: c=" + c);
         super.setBackground(c);
      }
   };
```

   При каждом вызове метода `setBackground()` будет генерироваться регистрационное сообщение. Как выяснить вызывающий метод и сформировать трассировку стека, описывается ниже.
5. С помощью метода `printStackTrace` из класса `Throwable` можно получить трассировку стека для любого исключения. Приведенный ниже код перехватывает исключение, отображает объект исключения и трассировку стека, а затем повторно генерирует исключение, чтобы найти предназначенный для него обработчик.

```
try
{
   ...
}
catch (Throwable t)
{
   t.printStackTrace();
   throw t;
}
```

   Для трассировки стека не нужно даже перехватывать исключение. Просто включите следующее выражение в любое место кода:

```
Thread.dumpStack();
```

6. Обычно трассировка кода выводится в поток `System.err`. С помощью метода `void PrintStackTrace(PrintWriter s)` ее результаты можно записать в файл. При желании результаты трассировки стека можно вывести в окне, предварительно преобразовав в строку.

```
StringWriter out = nw StringWriter();
new Throwable().printStackTrace(new PrintWriter(out));
String trace = out.toString;
```

(Классы PrinterWriter и StringWriter описываются в главе 12.)

7. Ошибки, возникающие при выполнении программы, удобно записывать в файл. Однако обычно ошибки посылаются в поток System.err, а не System.out. Следовательно, нельзя вывести ошибки в файл с помощью обычной команды

```
java MyProgram > errors.txt
```

В системе Unix, а также в Windows NT/2000/XP это – не проблема. Например, если вы работаете с оболочкой bash, перехватите поток ошибок с помощью приведенной ниже команды.

```
java MyProgram 2> errors.txt
```

Для того чтобы направить оба потока, Stream.err и Stream.out, в один и тот же файл, используйте команду

```
java MyProgram 2>&1 errors.txt
```

Некоторые операционные системы (например, Windows 95/98/Me) не имеют такого удобного механизма. В этой ситуации нужно воспользоваться следующей программой на языке Java:

```
import java.io.*;
public class Errout
{
   public static void main(String[] args) throws IOException
   {
      Process p = Runtime.getRuntime().exec(args);
        BufferedReader err = new BufferedReader(new
         InputStreamReader(p.getErrorStream()));
      String line;
      while ((line = err.readLine()) != null)
         System.out.println(line);
   }
}
```

Затем запустите эту программу таким образом:

```
java Errour java MyProgram.java > errors.txt
```

В системе Windows тот же результат можно получить более эффективным способом, скомпилировав следующую программу в файл errout.exe:

```
#include <io.h>
#include <stdio.h>
#include <process.h>
int main(int argc, char* argv[])
{
   dup2(1,2); /* Переключаем поток stderr на поток stdout. */
   execvp(argv[1], argv + 1);
   return 0;
}
```

Затем этот файл можно запустить с помощью команды

```
errout java MyProgram.java > errors.txt
```

8. Для того чтобы отследить загрузку класса, запустите виртуальную машину Java с опцией -verbose. На экране появятся такие строки:

```
[Opened /usr/local/jdk5.0/jre/lib/rt.jar]
[Opened /usr/local/jdk5.0/jre/lib/jsse.jar]
[Opened /usr/local/jdk5.0/jre/lib/jce.jar]
[Opened /usr/local/jdk5.0/jre/lib/charsets.jar]
[Loaded java.lang.Object from shared objects file]
[Loaded java.io.Serializable from shared objects file]
[Loaded java.lang.Comparable from shared objects file]
[Loaded java.lang.CharSequence from shared objects file]
[Loaded java.lang.String from shared objects file]
[Loaded java.lang.reflect.GenericDeclaration from shared  objects file]
[Loaded java.lang.reflect.Type from shared objects file]
[Loaded java.lang.reflect.AnnotatedElement from shared objects file]
[Loaded java.lang.Class from shared objects file]
[Loaded java.lang.Cloneable from shared objects file]
```

Этот способ оказывается полезным для диагностики ошибок, связанных с поиском классов.

9. Любуясь окнами из библиотеки Swing и удивляясь, как красиво и ровно расположены компоненты, естественно поинтересоваться, как именно они устроены. Для этого нужно нажать клавиши <Ctrl+Shift+F1>, и все компоненты будут выведены на экран по порядку.

```
FontDialog[frame0,0,0,300x200,layout=java.awt.BorderLayout,...
 javax.swing.JRootPane[,4,23,292x173,layout=javax.swing.
     JRootPane$RootLayout,...
   javax.swing.JPanel[null.glassPane,0,0,292x173,hidden,
      layout=java.awt.FlowLayout,...
   javax.swing.JLayeredPane[null.layeredPane,0,0,292x173,...
    javax.swing.JPanel[null.contentPane,0,0,292x173,layout=java.
        awt.GridBagLayout,...
     javax.swing.JList[,0,0,73x152,alignmentX=null,
         alignmentY=null,...
       javax.swing.CellRendererPane[,0,0,0x0,hidden]
        javax.swing.DefaultListCellRenderer$UIResource[,-73,-19,0x0,...
     javax.swing.JCheckBox[,157,13,50x25,layout=javax.swing.
        OverlayLayout,...
     javax.swing.JCheckBox[,156,65,52x25,layout=javax.swing.
        OverlayLayout,...
     javax.swing.JLabel[,114,119,30x17,alignmentX=0.0,
        alignmentY=null,...
     javax.swing.JTextField[,186,117,105x21,alignmentX=null,
        alignmentY=null,...
     javax.swing.JTextField[,0,152,291x21,alignmentX=null,
        alignmentY=null,...
```

10. Если разработанный вами собственный компонент пользовательского интерфейса на экране отображается некорректно, вы оцените удобство отладчика из пакета Swing. Даже если вы никогда не создавали свои собственные компоненты, очень поучительно проследить, как именно компоненты отображаются на экране. Чтобы переключиться на отладку компонента библиотеки Swing, нужно вызвать метод `setDebugGraphicsOptions()` класса `JComponent`. Этот метод имеет такие опции.

`DebugGraphics.FLASH_OPTION`	Перед тем, как отобразить линию, прямоугольник и текст на экране, показывает их в кадре
`DebugGraphics.LOG_OPTION`	Выводит на экран сообщение о каждой графической операции
`DebugGraphics.BUFFERED_OPTION`	Отображает операции, которые выполняются в видеобуфере
`DebugGraphics.NONE_OPTION`	Отключает отладку графики

Для того чтобы описанный выше механизм работал правильно, необходимо отключить опцию двойной буферизации, которая служит для подавления мерцания экрана при обновлении окна. Это можно сделать с помощью выражений, приведенных ниже.

```
RepaintManager.currentManager(getRootPane()).
   setDoubleBufferingEnabled(false);
((JComponent)getContentPane()).
   setDebugGraphicsOptions(DebugGraphics.FLASH_OPTION);
```

Поместите эти строки в конец конструктора вашего фрейма. При выполнении программы вы увидите панель, которая медленно заполняется содержимым. Для локальной отладки каждого компонента воспользуйтесь методом `setDebugGraphicsOptions()`. Продолжительность, количество и цвет кадров можно регулировать (подробности – в описании класса `DebugGraphics` в документации).

11. В JDK 5.0 была добавлена опция компилятора `-Xlint`, предназначенная для решения проблем, связанных с кодом программы. Например, если вы вызовете компилятор так, как указано ниже, то он проинформирует вас о пропущенных операторах `break` в блоках `switch`.

```
javac -Xlint:fallthrough
```

Термин `lint` ранее применялся для описания инструмента, предназначенного для обнаружения потенциальных проблем в программах, написанных на C. Теперь он обозначает все инструменты, отмечающие языковые конструкции, которые могут быть некорректными. Ниже описаны допустимые значения опции `-Xlint`.

`-Xlint` или `-Xlint:all`	Выполняет все проверки
`-Xlint:deprecation`	То же, что и `-deprecation`, т.е. выполняется проверка на наличие методов, не рекомендованных к применению
`-Xlint:fallthrough`	Проверяет наличие операторов `break` в блоках `switch`
`-Xlint:finally`	Предупреждает о наличии выражений `finally`, которые не могут нормально завершиться
`-Xlint:none`	Проверки не выполняются
`-Xlint:path`	Проверяет, существуют ли каталоги, указанные в пути к классам и исходным файлам
`-Xlint:serial`	Предупреждает о сериализуемых классах без `serialVersionUID` (подробно этот вопрос будет рассмотрен в главе 12)

`-Xlint:unchecked`	Предупреждает о сомнительных преобразованиях между универсальными и низкоуровневыми типами

12. В JDK 5.0 была реализована поддержка мониторинга и управления Java-приложениями. В виртуальной машине можно установить агентов, которые будут отслеживать расход памяти, использование потоков, загрузку классов и т.п. Эта возможность важна для работы с программами большого объема, работающими в течение длительного времени, например с серверами приложений. Для демонстрации новых возможностей в комплект поставки JDK включен графический инструмент `jconsole`, который отображает статистические данные о производительности виртуальной машины (рис. 11.4). Для того чтобы включить мониторинг, надо указать при запуске виртуальной машины опцию `-Dcom.sun.management.jmxremote`. Затем следует найти в операционной системе идентификатор процесса, поддерживающего работы виртуальной машины. В Unix/Linux для этой цели надо использовать утилиту `ps`, а в системе Windows – диспетчер задач. Затем можно запускать программу `jconsole`.

```
java -Dcom.sun.management.jmxremote MyProgram.java
jconsole processID
```

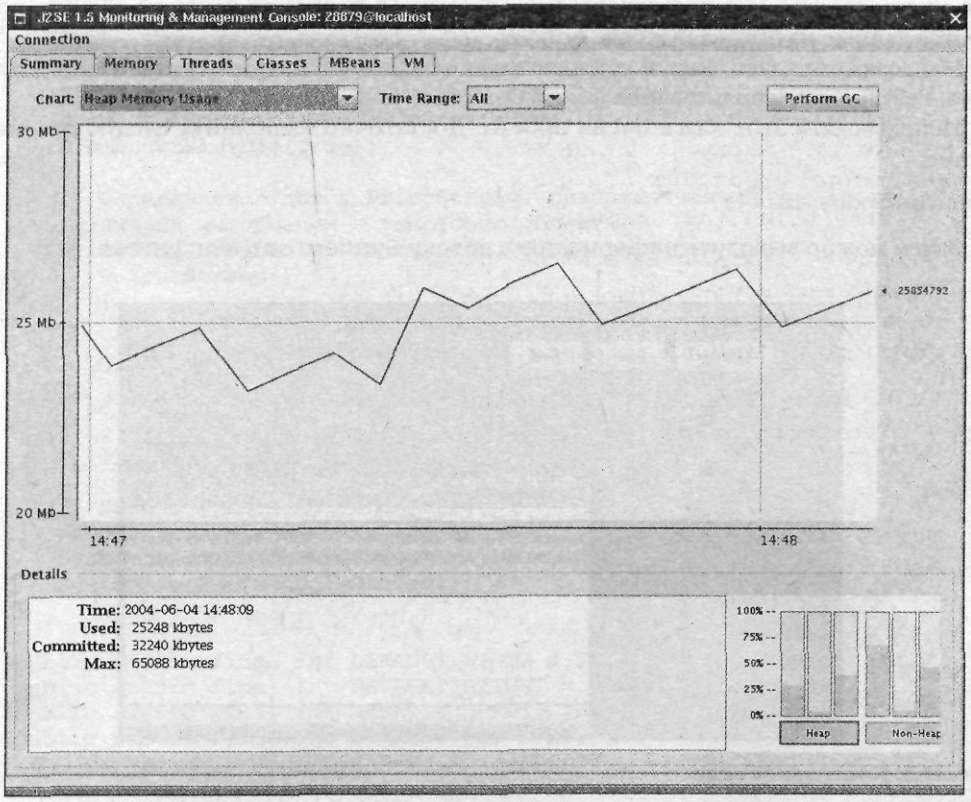


Рис. 11.4. Окно программы `jconsole`

13. Если запустить интерпретатор языка Java с опцией `-Xprof`, он запустит устаревший *профилировщик* (profiler), определяющий наиболее часто используемые методы. Профилировщик выводит информацию в поток `System.out`. В этом потоке также будет содержаться информация о том, какие методы были скомпилированы с помощью синхронной компиляции.

Опции компилятора, начинающиеся с `-X`, официально не поддерживаются и в некоторых версиях JDK могут отсутствовать. Получить список нестандартных опций можно, выполнив команду `java -X`.

Использование консольного окна

Если вы запускаете аплет в браузере, то не увидите сообщений, передаваемых в поток `System.out`. Большинство браузеров поддерживают разновидности консольных окон. (Обратитесь к справочной системе вашего браузера.) Например, браузеры Netscape Navigator, Internet Explorer 4 и последующие версии имеют консольное окно. Если вы работаете с программой Java Plug-In, проверьте флажок опции Show Java Console на панели конфигурации (см. главу 10).

Окно Java Console снабжено полосами прокрутки, так что в нем можно просматривать довольно пространные сообщения. Пользователи Windows имеют несомненное преимущество над пользователями системы DOS, в окне которой обычно появляются данные из потока `System.out`.

Мы создадим класс окна, в котором можно будет просматривать сообщения отладчика. Результаты использования класса `ConsoleWindow` показаны на рис. 11.5.

Использовать этот класс очень просто. Достаточно выполнить следующее выражение:

```
ConsoleWindow.init();
```

Затем можно выводить информацию в потоки `System.out` или `System.err`.

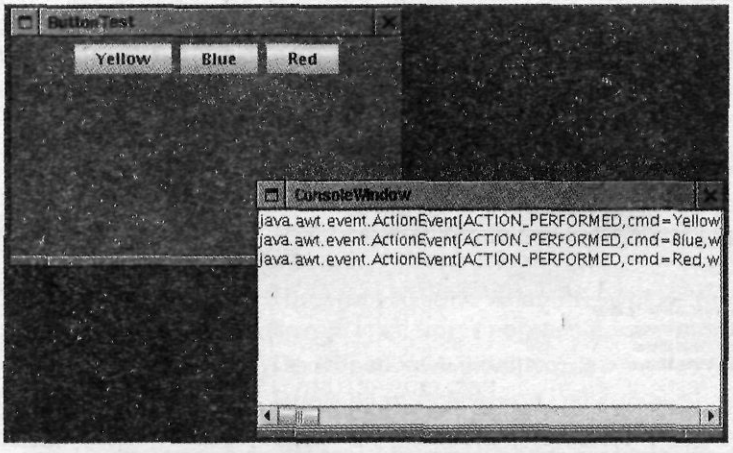


Рис. 11.5. Консольное окно

В листинге 11.5 показан код класса ConsoleWindow. Как видим, этот класс очень прост. Сообщения отображаются в текстовой области JTextArea на панели JScrollPane. Для переключения потока вывода или потока ошибок на особый поток, выводящий сообщения в текстовую область, вызываются методы System.setOut() и System.setErr().

Листинг 11.5. Содержимое файла ConsoleWindow.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;

/**
   Окно, предназначенное для отображения информации,
   System.out записанной в потоки и System.err.
*/
public class ConsoleWindow
{
   public static void init()
   {
      JFrame frame = new JFrame();
      frame.setTitle("ConsoleWindow");
      final JTextArea output = new JTextArea();
      output.setEditable(false);
      frame.add(new JScrollPane(output));
      frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      frame.setLocation(DEFAULT_LEFT, DEFAULT_TOP);
      frame.setFocusableWindowState(false);
      frame.setVisible(true);

      // Определяется поток PrintStream, предназначенный для
      // вывода информации в текстовую область.
      PrintStream consoleStream = new PrintStream(new
         OutputStream()
         {
            public void write(int b) {} // Не вызывается.
            public void write(byte[] b, int off, int len)
            {
               output.append(new String(b, off, len));
            }
         });

      // Данные System.out и System.err
      // будут передаваться в новый поток
      System.setOut(consoleStream);
      System.setErr(consoleStream);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
   public static final int DEFAULT_LEFT = 200;
   public static final int DEFAULT_TOP = 200;
}
```

Отслеживание событий AWT

Создавая пользовательский интерфейс на языке Java, мы должны знать, какие события происходят и каким компонентам они передаются. К сожалению, в документации AWT об этом говорится мало. Предположим, что мы хотим выводить подсказку в строке состояния, в зависимости от того, в какой части экрана перемещается курсор мыши. Библиотека AWT генерирует события, связанные с мышью и фокусом, которые можно перехватить.

Рассмотрим полезный класс `EventTracer`, предназначенный для слежения за этими событиями. Он выводит все методы, обрабатывающие события, а также их параметры. На рис. 11.6 показан результат отслеживания событий.

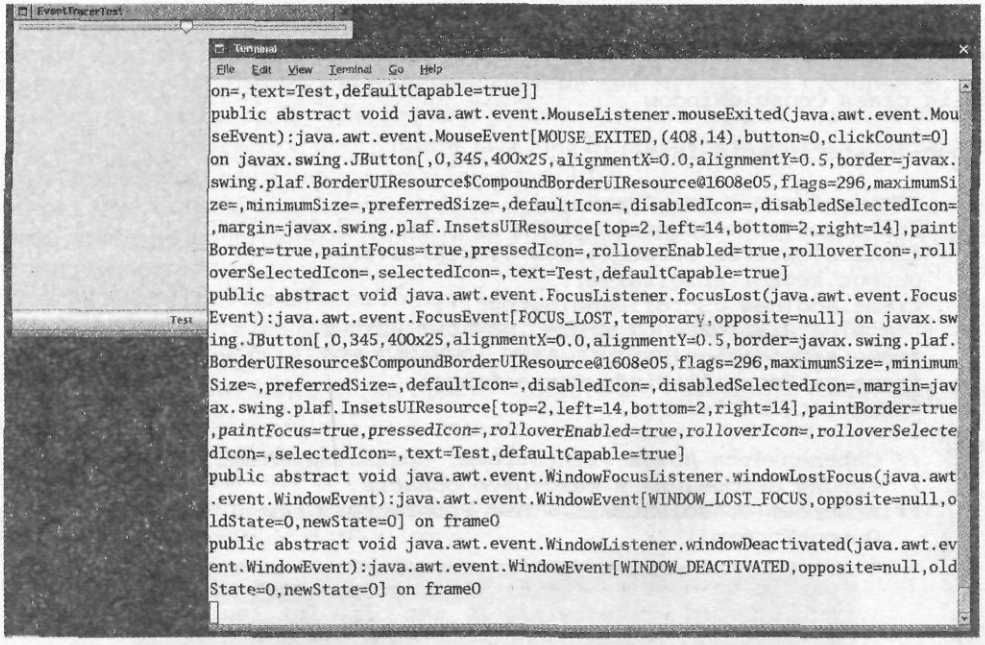


Рис. 11.6. Результат работы класса `EventTracer`

Для того чтобы разобраться в этих сообщениях, добавьте компонент, события которого вы хотите отслеживать, в объект класса `EventTracer`.

```
EventTracer tracer = new EventTracer();
tracer.add(frame);
```

Он создает текстовое описание всех событий.

```
public abstract void
java.awt.event.MouseListener.mouseExited(java.awt.event.MouseEvent):
java.awt.event.MouseEvent[MOUSE_EXITED,(408,14),button=0,clickCount=0] on
javax.swing.JButton[,0,345,400x25,...]
public abstract void
java.awt.event.FocusListener.focusLost(java.awt.event.FocusEvent):
java.awt.event.FocusEvent[FOCUS_LOST,temporary,opposite=null] on
javax.swing.JButton[,0,345,400x25,...]
```

Эти сообщения можно отобразить в консольном окне или записать в файл.

Код класса `EventTracer` приведен в листинге 11.6. Идея, лежащая в основе этого класса, довольно проста, хотя его реализация выглядит немного необычно.

1. Когда вы добавляете некий компонент в объект класса `EventTracer` с помощью метода `add()`, интроспективные средства JavaBeans ищут в нем методы, имеющие вид `void addXxxListener(XxxEvent)`. (Информацию о JavaBeans можно найти в главе 7 второго тома.) Для каждого найденного метода генерируется объект `EventSetDescriptor`. Каждый экземпляр этого класса передается методу `addListener()`.
2. Если компонент является контейнером, следует перечислить вложенные в него компоненты и рекурсивно вызвать метод `add()` для каждого из них.
3. Методу `addListener()` передаются два параметра: компонент, за событиями которого мы следим, и объект `EventSetDescriptor`. Метод `getListenerType()` класса `EventSetDescriptor` возвращает объект `Class`, описывающий интерфейс обработчика события, например `ActionListener` или `ChangeListener`. Для интерфейса создается proxy-объект. Он выводит имя и параметр вызванного метода события. Метод `getAddListenerMethod()` класса `EventSetDescriptor` возвращает объект `Method`. Этот объект используется для добавления proxy-объекта в компонент, в котором он выступает в качестве обработчика события.

   Эта программа представляет собой хорошую иллюстрацию возможностей механизма отражения. Мы можем не знать, что в классе `JButton` есть метод `addActionListener()`, а в классе `Jslider` — метод `addChangeListener()`. Механизм отражения выявит эти факты за нас.

Механизм proxy значительно упрощает программу. В предыдущих изданиях этой книги нам приходилось определять обработчик, который одновременно реализовывал интерфейсы `MouseListener`, `ComponentListener`, `FocusListener`, `KeyListener`, `Listener`, `ActionListener` и `ItemListener`, а также несколько десятков методов, выводящих соответствующее событие. О механизме `proxy` см. в главе 6.

В листинге 11.7 приведена программа, иллюстрирующая работу класса `EventTracer`. Эта программа выводит на экран фрейм с кнопкой и линейным регулятором и отслеживает события, генерируемые этими компонентами.

Листинг 11.6. Содержимое файла `EventTracer.java`

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.lang.reflect.*;

public class EventTracer
{
   public EventTracer()
   {
      // Обработчик всех proxy.
      handler = new
         InvocationHandler()
```

```
      {
         public Object invoke(Object proxy,
            Method method, Object[] args)
         {
            System.out.println(method + ":" + args[0]);
            return null;
         }
      };
}

/**
   Добавляет объекты EventTracer для всех событий,
   которые можно прослушивать у данного компонента и его
   дочерних классов.
   @param c Компонент
*/
public void add(Component c)
{
   try
   {
      // Получение всех событий, которые можно обработать.

      BeanInfo info = Introspector.getBeanInfo(c.getClass());

      EventSetDescriptor[] eventSets =
         info.getEventSetDescriptors();
      for (EventSetDescriptor eventSet : eventSets)
         addListener(c, eventSet);
   }
   catch (IntrospectionException e) {}
   // Если генерируется исключение,
   // обработчики не добавляются.


   if (c instanceof Container)
   {
      // Получение всех дочерних объектов
       // и рекурсивный вызов метода add.

      for (Component comp : ((Container) c).getComponents())
         add(comp);
   }
}

/**
   Добавление обработчика для заданного множества событий.
   @param c Компонент
   @param eventSet Описание интерфейса обработчика.
*/
public void addListener(Component c,
   EventSetDescriptor eventSet)
{
   // Создание proxy-объекта для данного типа обработчика
   // и перенаправление вызовов.
   Object proxy = Proxy.newProxyInstance(null,
      new Class[] { eventSet.getListenerType() }, handler);
```

```
      // Добавление proxy в качестве обработчика для компонента.
      Method addListenerMethod = eventSet.getAddListenerMethod();
      try
      {
         addListenerMethod.invoke(c, proxy);
      }
      catch(InvocationTargetException e) {}
      catch(IllegalAccessException e) {}
      // Если генерируется исключение,
      // обработчики не добавляются.
   }

   private InvocationHandler handler;
}
```

Листинг 11.7. Содержимое файла `EventTracerTest.java`

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class EventTracerTest
{
   public static void main(String[] args)
   {
      JFrame frame = new EventTracerFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

class EventTracerFrame extends JFrame
{
   public EventTracerFrame()
   {
      setTitle("EventTracerTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление линейного регулятора и кнопки
      add(new JSlider(), BorderLayout.NORTH);
      add(new JButton("Test"), BorderLayout.SOUTH);

      // Перехват всех событий, связанных с компонентами
      // в составе фрейма.
      EventTracer tracer = new EventTracer();
      tracer.add(this);
   }

   public static final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 400;
}
```

Класс Robot

В версии 1.3 языка Java 2 появился класс `Robot`, *который можно использовать для* передачи нажатий клавиш и щелчков мыши в любую AWT-программу. Этот класс предназначен для автоматической проверки пользовательского интерфейса.

Для того чтобы создать объект класса `Robot`, нужно сначала получить объект класса `GraphicsDevice`. Устройство экрана, заданное по умолчанию, можно определить с помощью следующей последовательности вызовов:

```
GraphicsEnvitonment environment
   = GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice screen
   = environment.getDefaultScreenDevice();
```

Затем создается объект класса `Robot`.

```
Robot robot = new Robot(screen);
```

Чтобы передать нажатие клавиш в программу, объект `Robot` *должен имитировать* нажатие и освобождение клавиш.

```
robot.keyPress(KeyEvent.VK_TAB);
robot.keyRelease(KeyEvent.VK_TAB);
```

Для имитации щелчка мыши сначала нужно переместить ее курсор, а затем нажать и отпустить кнопку.

```
robot.mouseMove(x, y);  // x и y - абсолютные координаты экрана
robot.mousePress(InputEvent.BUTTON1_MASK);
robot.mouseRelease(InputEvent.BUTTON1_MASK);
```

Основная идея проверки состоит в том, что, имитируя нажатие клавиш и щелчки мыши, можно увидеть, правильно ли приложение реагирует на них. Затем нужно захватить содержимое экран с помощью метода `createScreenCapture()`.

```
Rectangle rect = new Rectangle(x, y, width, height);
BufferedImage image = robot.createScreenCapture(rect);
```

Координаты, задаваемые в конструкторе `Rectangle`, также являются абсолютными координатами экрана и выражаются в пикселях.

Кроме того, чтобы приложение могло перехватить команды, выполняемые объектами класса `Robot`, между ними нужны небольшие задержки. Для этого предназначен метод `delay()`, при вызове которого указывается величина задержки в миллисекундах.

```
robot.delay(1000); // Задержка на 1000 миллисекунд.
```

Программа, приведенная в листинге 11.8, иллюстрирует применение объектов класса `Robot`. Эти объекты тестируют программу для работы с кнопками, рассмотренную нами в главе 8. Сначала в ответ на нажатие клавиши пробела активизируется левая кнопка. Затем объект `Robot` ожидает две секунды, чтобы дать пользователю возможность увидеть состояние экрана. По истечении этого срока объект класса `Robot` имитирует нажатие клавиши <Tab> и клавиши пробела для активизации следующей кнопки. После этого имитируется щелчок мышью на третьей кнопке. (Для того чтобы действительно щелкнуть на этой кнопке, в программе нужно правильно задать параметры `x` и `y`.) В заключение содержимое экрана перехватывается и отображается в отдельном фрейме (рис. 11.7).

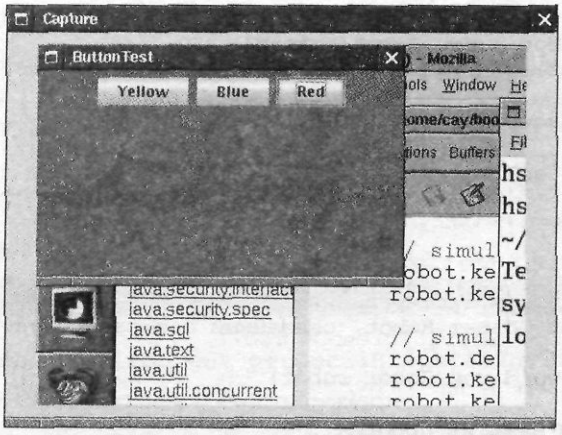


Рис. 11.7. Захват содержимого экрана с помощью объекта Robot

Этот пример показывает, что класс Robot сам по себе для тестирования пользовательского интерфейса не совсем удобен. Скорее, он представляет собой базовые "строительные блоки", из которых можно сконструировать средство для тестирования программ. Профессиональные средства тестирования могут фиксировать, хранить и воспроизводить сценарии поведения пользователя, а также определять расположение компонентов на экране, чтобы не щелкать мышью наугад. В момент написания книги класс Robot был новинкой, и нам пока неизвестны средства для тестирования пользовательского интерфейса, которые были бы созданы с его помощью. Надеемся, в будущем они все же появятся.

Листинг 11.8. Содержимое файла RobotTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;

public class RobotTest
{
   public static void main(String[] args)
   {
      // Создание фрейма, содержащего панель с кнопками.

      ButtonFrame frame = new ButtonFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);

      // Связывание объекта Robot с экранным устройством.

      GraphicsEnvironment environment =
          GraphicsEnvironment.getLocalGraphicsEnvironment();
      GraphicsDevice screen =
          environment.getDefaultScreenDevice();

      try
```

```
      {
         Robot robot = new Robot(screen);
         run(robot);
      }
      catch (AWTException e)
      {
         e.printStackTrace();
      }
   }

   /**
      Запуск процедуры тестирования.
      @param robot Объект Robot, связанный с экранным устройством.
   */
   public static void run(Robot robot)
   {
      // Имитация нажатия пробела.
      robot.keyPress(' ');
      robot.keyRelease(' ');

      // Имитация нажатия клавиши <Tab> с последующим
      // нажатием клавиши пробела.
      robot.delay(2000);
      robot.keyPress(KeyEvent.VK_TAB);
      robot.keyRelease(KeyEvent.VK_TAB);
      robot.keyPress(' ');
      robot.keyRelease(' ');

      // Имитация щелчка правой кнопкой мыши.
      robot.delay(2000);
      robot.mouseMove(200, 50);
      robot.mousePress(InputEvent.BUTTON1_MASK);
      robot.mouseRelease(InputEvent.BUTTON1_MASK);

      // Захват содержимого экрана и вывод
      // полученного изображения.
      robot.delay(2000);
      BufferedImage image = robot.createScreenCapture(new
         Rectangle(0, 0, 400, 300));

      ImageFrame frame = new ImageFrame(image);
      frame.setVisible(true);
   }
}

/**
   Фрейм для вывода захваченного изображения.
*/
class ImageFrame extends JFrame
{
   /**
      @param image Изображение для вывода
   */
   public ImageFrame(Image image)
   {
      setTitle("Capture");
```

```
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        JLabel label = new JLabel(new ImageIcon(image));
        add(label);
    }

    public static final int DEFAULT_WIDTH = 450;
    public static final int DEFAULT_HEIGHT = 350;
}
```

API java.awt.GraphicsEnvironment 1.2

- `static GraphicsEnvironment getLocalGraphicsEnvironment()`

  Возвращает локальное графическое окружение.

- `GraphicsDevice getDefaultScreenDevice()`

  Возвращает экранное устройство, заданное по умолчанию. Обратите внимание на то, что компьютеры с несколькими мониторами имеют по одному устройству на каждый экран, поэтому, чтобы получить массив, содержащий все устройства, используйте метод `getScreenDevices`.

API java.awt.Robot 1.3

- `Robot(GraphicsDevice device)`

  Создает объект класса `Robot`, который может взаимодействовать с данным устройством.

- `void keyPress(int key)`
- `void keyRelease(int key)`

  Имитирует нажатие и отпускание клавиши.

Параметр	`key`	Код клавиши. Подробную информацию о кодах клавиш можно найти, рассмотрев класс `KeyStroke`

- `void mouseMove(int x, int y)`

  Имитирует перемещение курсора мыши.

Параметр	`x, y`	Абсолютные координаты курсора мыши, выраженные в пикселях

- `void mousePress(int eventMask)`
- `void mouseRelease(int eventMask)`

  Имитирует нажатие и отпускание кнопки мыши.

Параметр	`eventMask`	Маска события, описывающая кнопку мыши. Дополнительную информацию о масках событий можно получить, рассмотрев класс `InputEvent`

- `void delay(int milliseconds)`

  Приостанавливает работу объекта `Robot` на заданное количество миллисекунд.

- `BufferedImage createScreenCapture(rectangle rect)`

  Захватывает часть экрана.

Параметр	`rect`	Часть экрана, подлежащая захвату. Задается в *абсолютных координатах*

Применение отладчика

Отладка программы с помощью вывода на экран промежуточных результатов — не самое приятное занятие. Вам приходится постоянно добавлять и удалять операторы, а затем компилировать программу заново. Отладчик намного облегчит вашу работу, поскольку в нем можно установить *точку останова* (breakpoint). Вплоть до этой точки программа выполняется без задержек, а после того, как выполнение программы приостанавливается, можно получить интересующие сведения о состоянии кода.

Отладчик JDB

В пакет JDK входит программа JDB — чрезвычайно старый отладчик, работающий в режиме командной строки. Интерфейс этой программы настолько примитивен, что пользоваться им можно лишь в крайнем случае. Действительно, это лишь иллюстрация возможностей отладки, а не профессиональная программа. Тем не менее мы кратко опишем ее, поскольку лучше иметь такой отладчик, чем не иметь никакого. Разумеется, во многих профессиональных средах программирования Java есть более удобные отладчики. Основные принципы работы всех отладчиков одинаковы, поэтому примеры из этого раздела можно использовать в процессе обучения.

В листингах 11.9–11.11 показана программа `ButtonTest` из главы 8, в код которой внесены некоторые ошибки. (Мы разделили программу на части и поместили каждый класс в отдельный файл, так как в некоторых отладчиках при одновременной работе с несколькими классами возникают проблемы.)

После щелчка на любой кнопке ничего не произойдет. Посмотрите на код — предполагается, что после щелчка на кнопке фон должен окраситься в соответствующий цвет.

Листинг 11.9. Содержимое файла `BuggyButtonTest.java`

```
import javax.swing.*;

public class BuggyButtonTest
{
  public static void main(String[] args)
  {
    BuggyButtonFrame frame = new BuggyButtonFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

Листинг 11.10. Содержимое файла `BuggyButtonFrame.java`

```
import java.awt.*;
import javax.swing.*;

public class BuggyButtonFrame extends JFrame
{
```

```
   public BuggyButtonFrame()
   {
      setTitle("BuggyButtonTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление панели к фрейму.

      BuggyButtonPanel panel = new BuggyButtonPanel();
      add(panel);
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
}
```

Листинг 11.11. Содержимое файла BuggyButtonPanel.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class BuggyButtonPanel extends JPanel
{
   public BuggyButtonPanel()
   {
      ActionListener listener = new ButtonListener();

      JButton yellowButton = new JButton("Yellow");
      add(yellowButton);
      yellowButton.addActionListener(listener);

      JButton blueButton = new JButton("Blue");
      add(blueButton);
      blueButton.addActionListener(listener);

      JButton redButton = new JButton("Red");
      add(redButton);
      redButton.addActionListener(listener);
   }

   private class ButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         String arg = event.getActionCommand();
         if (arg.equals("yellow"))
            setBackground(Color.yellow);
         else if (arg.equals("blue"))
            setBackground(Color.blue);
         else if (arg.equals("red"))
            setBackground(Color.red);
      }
   }
}
```

Ошибку в такой короткой программе можно обнаружить и без отладчика, для этого достаточно внимательно прочитать код. Допустим, что наша программа очень сложна. В этой ситуации нам нужен отладчик.

Для того чтобы применить отладчик JDB, надо сначала скомпилировать программу с опцией -g.

```
javac -g BuggyButtonTest.java BuggyButtonFrame.java
   BuggyButtonPane.java
```

В этом случае компилятор добавляет в файлы классов имена локальных переменных и другую отладочную информацию. После этого следует запустить отладчик.

```
jdb BuggyButtontest
```

На экране появится что-то наподобие следующей строки:

```
Initializing jdb...
>
```

Приглашение > означает, что отладчик ожидает команду. В табл. 11.4 приведены все команды отладчика. Элементы, заключенные в квадратные скобки [...], являются необязательными, а при наличии суффикса (s) можно задавать несколько параметров, разделяемых пробелами.

Таблица 11.4. Команды отладчика

`threads [`*`группа_потоков`*`]`	Перечисляет потоки
`thread` *`номер_потока`*	Задает поток по умолчанию
`suspend [`*`номер(а)_потока(ов)`*`]`	Приостанавливает работу потоков (по умолчанию: `all`)
`resume [`*`номер(а)_потока(ов)`*`]`	Возобновляет работу потоков (по умолчанию: `all`)
`where [`*`номер(а)_потока(ов)`*`]` или `all`	Выводит содержимое стека потока
`wherei [`*`номер(а)_потока(ов)`*`]` или `all`	Выводит содержимое стека потока и счетчик программы
`threadgroups`	Перечисляет группы потоков
`threadgroup` *`имя`*	Задает текущую группу потоков
`print` *`имя(имена)`*	Выводит информацию об объекте или поле
`dump` *`имя(имена)`*	Выводит информацию обо всех объектах
`locals`	Выводит значения всех локальных переменных
`classes`	Перечисляет известные в данный момент классы
`methods` *`класс`*	Перечисляет методы класса
`stop in` *`класс.метод`*	Устанавливает точку останова в методе
`stop at` *`класс:строка`*	Устанавливает точку останова в строке
`up [`*`n`*`]`	Перемещение вверх по стеку потока
`down [`*`n`*`]`	Перемещение вниз по стеку потока

Окончание табл. 11.4

`clear` *`класс:строка`*	Удаляет точку останова
`step`	Выполняет текущую строку, перемещаясь по строкам внутри класса
`stepi`	Выполняет текущую команду
`step up`	Выполняет все команды до конца текущего метода
`next`	Выполняет текущую строку, вызов метода считается одной командой
`cont`	Продолжает выполнение, начиная с точки останова
`catch` *`класс`*	Прерывает работу программы, генерируя указанное исключение
`ignore` *`класс`*	Игнорирует указанное исключение
`list` [*`строка`*]	Выводит текст программы
`use` [*`путь`*]	Отображает или изменяет путь к исходному файлу
`memory`	Информирует об использовании памяти
`gc`	Удаляет ненужные объекты
`load` *`класс`*	Загружает класс, предназначенный для отладки
`run` [*`класс`* [*`параметры`*]]	Начинает выполнение загруженного класса
`!!`	Повторяет последнюю команду
`help` (или `?`)	Перечисляет команды
`exit` (или `quit`)	Выход из отладчика

В этом разделе мы рассматриваем лишь наиболее полезные команды отладчика JDB. Основная идея проста: установить одну или несколько точек останова, а затем запустить программу. Когда программа достигает точки останова, ее выполнение прерывается. В этот момент можно проверить значения локальных переменных и сравнить их с ожидаемыми.

Для того чтобы установить точку останова, используется одна из приведенных ниже команд.

```
stop in класс.метод
stop at класс:строка
```

Например, установим точку останова в методе `actionPerformed()` класса `BuggyButtonTest`.

```
stop in BuggyButtonPanel$ButtonListener.actionPerformed
```

Теперь, для того, чтобы запустить программу и выполнить ее до точки останова, нужно выполнить следующую команду

```
run
```

Начнется выполнение программы, но точка останова никак не проявит себя, пока не начнется обработка кода в методе `actionPerformed()`. Чтобы это случилось, щелкните на кнопке Yellow. Отладчик прервет работу *в начале* метода `actionPerformed()`. На экране появится следующее сообщение:

```
Breakpoint hit: thread="AWT-EventQueue-
0",BuggyButtonPanel$ButtonListener.actionPerformed(). line=28, bci=0
  28          String arg = event.getActionCommand();
```

Команда `list` позволяет уточнить, в каком месте программы произошел останов. По этой команде отладчик выведет на экран текущую строку, а также несколько предыдущих и последующих строк. Перед строками указаны их номера.

```
24          private class ButtonListener implements ActionListener
25          {
26             public void actionPerformed(ActionEvent event)
27             {
28   =>           String arg = event.getActionCommand();
29               if (arg.equals("yellow"))
30                  setBackground(Color.yellow);
31               else if (arg.equals("blue"))
32                  setBackground(Color.blue);
33               else if (arg.equals("red"))
```

Для того чтобы увидеть все локальные переменные, введите команду `locals`.

```
Method arguments:
  event = instance of java.awt.event.ActionEvent(id=698)
Local variables:
```

Для получения более подробной информации используется команда `dump`.

```
dump переменные
```

Например, для отображения всех полей в переменной `event` нужно выполнить команду

```
dump event
```

Тогда на экране появится следующая информация:

```
event = instance of java.awt.event.ActionEvent(id=698) {
   SHIFT_MASK: 1
   CTRL_MASK: 2
   META_MASK: 4
   ALT_MASK: 8
   ACTION_FIRST: 1001
   ACTION_LAST: 1001
   ACTION_PERFORMED: 1001
   actionCommand: "Yellow"
   modifiers: 0
   serialVersionUID: -7671078796273832149
   ...
```

Для пошаговой отладки программы предназначены две команды. Команда `step` позволяет выполнять отладку, заходя внутрь вызываемых методов, а команда `next` выполняет переход к следующей строке, считая вызов метода одной командой. Дваж-

ды выполните команду next, а затем воспользуйтесь командой list, чтобы обнаружить точку программы, в который вы находитесь в текущий момент.

Выполнение программы остановлено на строке 31.

```
27            {
28               String arg = event.getActionCommand();
29               if (arg.equals("yellow"))
30                  setBackground(Color.yellow);
31  =>           else if (arg.equals("blue"))
32                  setBackground(Color.blue);
33            else if (arg.equals("red"))
34                  setBackground(Color.red);
35            }
```

Это не совсем то, чего мы хотели. Предполагалось, что будет вызван метод `setColor(Color.yellow)`.

Просмотрим содержимое переменной `arg`.

```
arg = "Yellow"
```

Теперь понятно, что произошло. Значение переменной `arg` *было равно* `Yellow` с прописной буквы Y, а сравнение

```
if (arg.equals("yellow"))
```

рассчитано на строчную букву y. Тайна раскрыта.

Для того чтобы окончить работу отладчика, выполните команду

```
quit
```

Этот пример показывает, что отладчик можно использовать для поиска ошибок, но интерфейс в режиме командной строки очень неудобен. Помните: если вы забыли, в какой точке находитесь в текущий момент, применяйте команды `list` и `locals`. Однако для отладки серьезных программ все же следует использовать более удобный отладчик.

Отладчик Eclipse

Интегрированная среда Eclipse предоставляет современный и удобный отладчик. С его помощью можно устанавливать точки останова, отслеживать значения переменных, запускать программу в пошаговом режиме и выполнять другие нужные действия.

Для того чтобы установить точку прерывания, переместите курсор на нужную строку и выберите пункт меню Run⇨Toggle Line Breakpoint или нажмите сочетание клавиш <Ctrl+F8>. Точка останова подсвечивается (рис. 11.8).

Чтобы начать отладку, выберите пункт меню Run⇨Debug As⇨Java Application. Программа начнет работу. Затем установите точку останова в первой строке метода `actionPerformed()`.

Когда отладчик остановится в выбранной точке, можно просмотреть стек вызовов и значения локальных переменных (рис. 11.9).

Чтобы выполнить один шаг кода, выберите пункт Run⇨Step into (<F5>) или Run⇨Step over (<F6>). В нашем примере достаточно дважды нажать клавишу <F6>, чтобы увидеть, как программа пропускает команду `setBackground(Color.yellow)`. Причину такого поведения можно обнаружить, изучив значение переменной `args` (рис. 11.10).

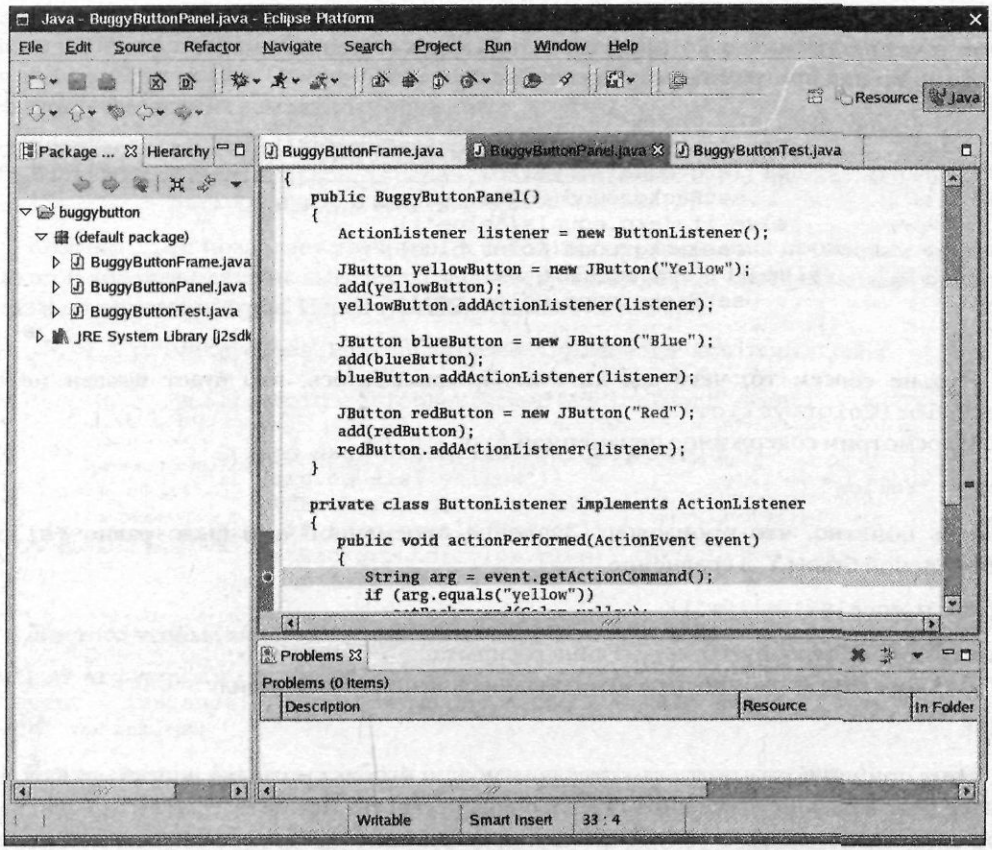


Рис. 11.8. Точка останова в отладчике Eclipse

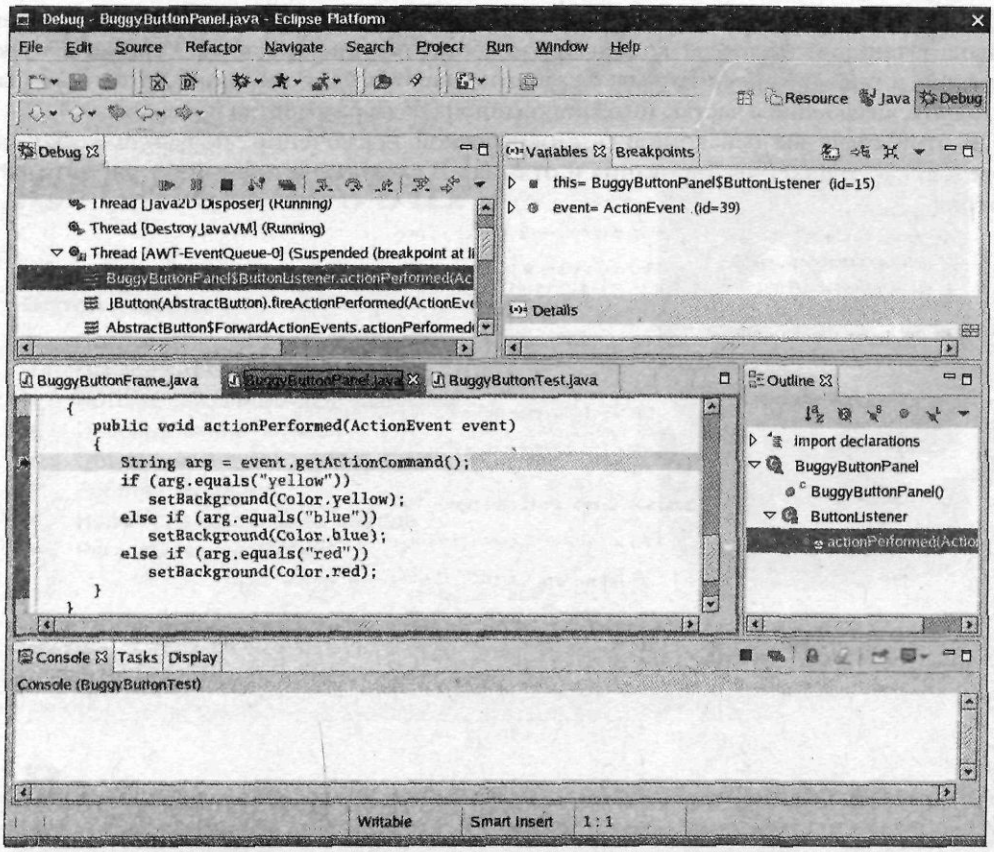


Рис. 11.9. В точке останова выполнение программы прерывается

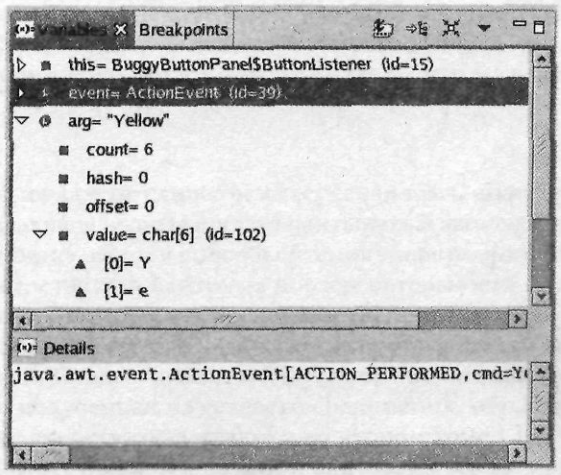


Рис. 11.10. Просмотр значений переменных

Как видим, отладчик Eclipse намного удобнее, чем JDB, поскольку он по умолчанию выводит текущий фрагмент кода и другую полезную информацию. Установка точки останова и проверка переменных также выполняются намного проще. Это типичный отладчик, являющийся частью интегрированной среды разработки программ.

В этой главе вы ознакомились с обработкой исключений, получили несколько советов, касающихся тестирования и отладки. В следующей главе будут рассмотрены потоки.

Глава 12

Потоки и файлы

В этой главе...

- Потоки
- Полное семейство потоков
- Потоки, связанные с ZIP-файлами
- Использование потоков
- Объектные потоки
- Работа с файлами
- Новые средства ввода-вывода
- Регулярные выражения

В этой главе излагаются методы работы с файлами и каталогами, а также способы считывания и записи информации. Кроме того, здесь описан механизм сериализации, позволяющий сохранять объекты так же легко, как текст или числовые данные. Далее мы рассмотрим новые возможности, реализованные в JDK 1.4. Глава завершается обсуждением регулярных выражений, хотя они не связаны с потоками и файлами. Мы не нашли более подходящего места для этой темы, впрочем, как и группа разработчиков языка Java, — в документации, сопровождающей пакет JDK 1.4, описание регулярных выражений находится в том же разделе, что и сведения о новых средствах ввода-вывода.

Потоки

Средства ввода-вывода — не слишком интересная тема, но изучить их все же нужно; программа, не обладающая возможностью считывать и записывать данные, никому не нужна. В этой главе описываются способы чтения информации из источника данных, который может предоставлять байтовую последовательность, и дальнейшей ее передачи любым адресатам, которые эту последовательность могут воспринять. Этими источниками и адресатами могут быть (и часто являются) файлы, сетевые соединения, а также блоки памяти. Следует иметь в виду, что, информация, размещенная в файлах, и информация, полученная из сетевого соединения, обрабатывается в основном одинаково. (Сетевые средства обсуждаются во втором томе.) Несмотря на то, что все данные в конечном итоге являются набором байтов, их все же удобнее представлять в виде некоторой структуры высокого уровня, например последовательности символов

или объектов. По этой причине мы лишь коснемся вопросов, связанных с низкоуровневым вводом-выводом, и сосредоточим внимание на средствах обмена данными высокого уровня.

В языке Java объект, из которого можно считывать последовательность байтов, называется *потоком ввода* (input stream), а объект, в который можно записывать эту последовательность данных, называется *потоком вывода* (output stream). Они определены в абстрактных классах `InputStream` и `OutputStream`. Потоки, ориентированные на байтовые последовательности, неудобны для обработки информации, записанной в формате Unicode. (Как вы помните, в Unicode кодовая единица содержит два байта). По этой причине классы для обработки символов в кодировке Unicode образуют отдельную иерархию, в основе которой лежат потоки `Reader` и `Writer`. Операции ввода и вывода, предусмотренные в этих классах, реализованы с учетом того, что символ представляется двумя байтами.

Абстрактные классы были описаны в главе 5. Напомним, что эти классы дают возможность выделять общие закономерности поведения классов и описывать их на более высоком уровне абстракции. Это позволяет создавать более ясные программы и облегчает анализ дерева наследования. Данный механизм используется и для обеспечения ввода-вывода в языке Java.

Как мы вскоре убедимся, из упомянутых четырех классов в языке Java формируется целое семейство, предназначенное для поддержки ввода-вывода.

Чтение и запись байтов

Класс `InputStream` содержит абстрактный метод

```
abstract int read()
```

Этот метод считывает и возвращает один байт. Обнаружив конец потока, этот метод возвращает число –1. Разработчик класса для конкретного потока ввода должен переопределить этот метод, реализовав необходимые функциональные возможности. Например, в классе `FileInputStream` этот метод считывает один байт из файла. Поток `System.in` является предопределенным объектом, представляющим собой экземпляр подкласса `InputStream`. Он позволяет вводить информацию с клавиатуры.

В классе `InputStream` есть и неабстрактные методы, предназначенные для чтения массива байтов, а также для пропуска заданного количества байтов при вводе. Эти методы вызывают абстрактный метод `read()`, поэтому в подклассах нужно переопределить только один метод.

Аналогично в классе `OutputStream` определен абстрактный метод

```
abstract void write(int b)
```

Этот метод позволяет записывать один байт в поток вывода.

Оба метода, `read()` и `write()`, могут *блокировать поток до тех пор, пока байт* не будет действительно считан или записан. Таким образом, если данные невозможно прочитать или записать немедленно (например, из-за загруженности сети), поток, содержащий вызов одного из этих методов, приостанавливается. Это дает другим потокам шанс выполнить за это время полезную работу. (Потоки рассматриваются во втором томе.)

Метод `available()` позволяет проверить байты, доступные для чтения в текущий момент. Благодаря использованию этого мтода приведенный ниже фрагмент кода вряд ли когда-нибудь заблокирует поток.

```
int bytesAvailable = in.available();
if (bytesAvailable > 0)
{
   byte[] data = new byte[bytesAvailable];
   in.read(data);
}
```

По окончании чтения или записи поток следует закрыть, вызвав метод `close()`. Этот вызов освобождает ресурсы операционной системы. Если приложение открывает слишком много потоков, не закрывая их, системные ресурсы могут оказаться исчерпанными. Закрытие потока вывода *принудительно выводит* (flush), или *сбрасывает*, данные из буфера, используемого этим потоком: все символы, которые временно накапливались в этом буфере, доставляются по назначению. В частности, если вы забыли закрыть файл, есть вероятность, что последний пакет байтов, записанный в буфере, никогда не будет доставлен. Данные из буфера можно сбросить и вручную, воспользовавшись методом `flush()`.

Даже если бы в классе потока были предусмотрены конкретные методы для работы с низкоуровневыми функциями чтения и записи, программисты редко бы использовали их, так как в программах не часто возникает необходимость считывать и записывать именно потоки байтов. Данные, которые интересуют пользователей, обычно представляют собой числа, строки и объекты.

В языке Java есть много классов потока, производных от основных классов `InputStream` и `OutputStream`. Эти классы позволяют работать с данными из файлов не на уровне байтов, а в том формате, в котором они обычно представляются.

API `java.io.InputStream 1.0`

- `abstract int read()`

  Читает из потока и возвращает байты. Если обнаружен конец потока, метод `read` возвращает число –1.

- `int read(byte[] b)`

  Читает данные в байтовый массив и возвращает количество фактически прочитанных байтов или число –1, если обнаружен конец потока. Максимальное количество байтов, которые может прочитать метод `read()`, определяется полем `b.length`.

- `int read(byte[] b, int off, int len)`

  Читает данные и помещает их в часть байтового массива. Возвращает фактическое количество прочитанных байтов или число –1, если обнаружен конец потока.

Параметры	`b`	Массив, в который записываются байты
	`off`	Смещение, определяющее позицию первого байта в массиве `b`
	`len`	*Максимально* возможное количество прочитанных байтов

- `long skip(long n)`

  Пропускает `n` байтов в потоке ввода. Возвращает фактическое количество пропущенных байтов (которое может быть меньше `n`, если обнаружен конец потока).

- `int available()`

  Возвращает количество байтов, доступных в потоке.

- `void close()`

  Закрывает поток ввода.

- `void mark(int readlimit)`

  Помещает маркер в текущую позицию входного потока. (Данную возможность поддерживают не все потоки.) Если из потока ввода прочитано больше, чем `readlimit` байтов, то маркер может быть потерян.

- `void reset()`

  Возврат к последнему маркеру. Последовательность вызовов метода `read()` приводит к повторному считыванию байтов. Если текущего маркера нет, текущая позиция в потоке не изменяется.

- `boolean markSupported()`

  Возвращает значение `true`, если поток допускает установку маркера.

`java.io.OutputStream 1.0`

- `abstract void write(int n)`

  Записывает байт.

- `void write(byte[] b)`

  Записывает все байты из массива `b`.

- `void write(byte[] b, int off, int len)`

  Записывает в поток часть массива `b`.

  Параметры `b` Массив, содержащий данные
  `off` Смещение, определяющее место первого байта в массиве `b`
  `len` Максимально возможное количество записанных байтов

- `void close()`

  Принудительно выводит остаток данных из буфера и закрывает поток вывода.

- `void flush()`

  Принудительно выводит остаток данных в буфере, посылая всю информацию по назначению.

Полное семейство потоков

В отличие от языка C, в котором есть только один тип `FILE*`, в языке Java существует целое семейство, состоящее более чем из 60 (!) типов потоков (рис. 12.1 и 12.2). Разработчики заявляют, что у них есть веская причина предоставить пользователям такой широкий выбор потоков: предполагается, что это снизит количество ошибок

программирования. Например, некоторые программисты, работающие на языке C, считают, что среди всех ошибок, связанных с вводом-выводом, чаще всего встречается попытка записать данные в файл, открытый только для чтения. (На самом деле это не самая распространенная ошибка.) Очевидно, что при этом выходные данные игнорируются. В языках Java и C++ компилятор распознает ошибки такого рода, поскольку в классе `InputStream` (Java) или `istream` (C++) нет методов, предназначенных для вывода данных.

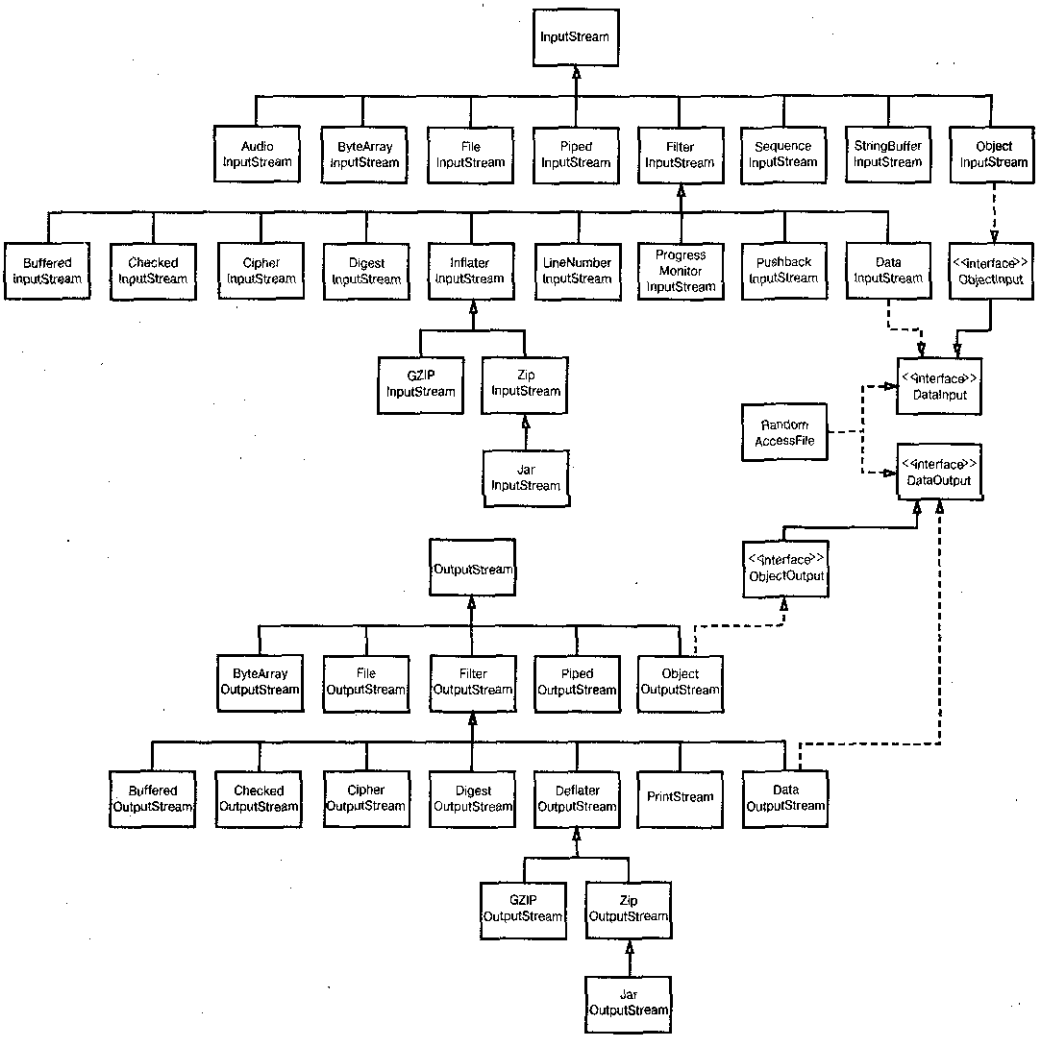


Рис. 12.1. Иерархия потоков ввода и вывода

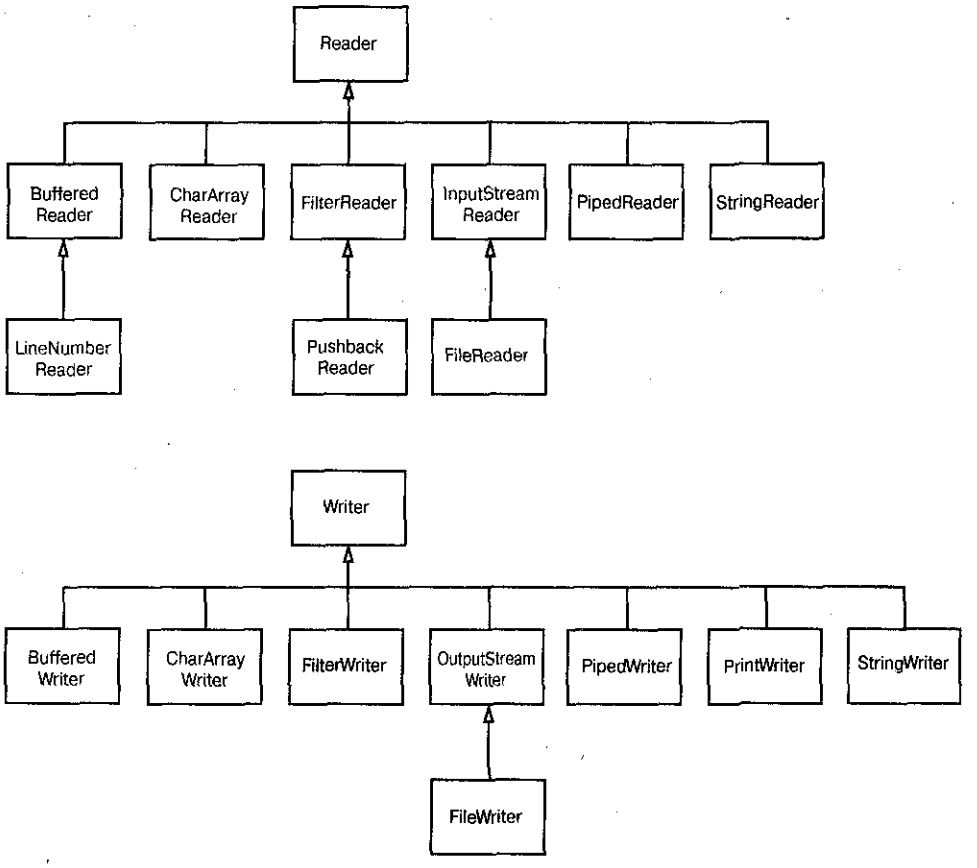


Рис. 12.2. Иерархия классов `Reader` и `Writer`

(На наш взгляд, в C++ и еще в большей степени в языке Java основной инструмент борьбы с ошибками — запугивание программистов. Крайняя сложность библиотек, предназначенных для работы с потоками, заставляет разработчиков быть чрезвычайно осторожными.)

Стандарт ANSI C++ предоставляет гораздо больше типов потоков, чем нужно обычному разработчику, например `isteam`, `ostream`, `iostream`, `ifstream`, `ofstream`, `fstream`, `wistream`, `wifstream`, `istrstream` и т.д (всего 18 классов). Java также перегружен потоками и предоставляет отдельные классы для буферизации, просмотра, произвольного доступа, форматирования текста и работы с двоичными данными.

Распределим членов этого семейства потоков по их назначению. Во главе стоят четыре абстрактных класса: `InputStream`, `OutputStream`, `Reader` и `Writer`. Мы не можем создавать экземпляры этих классов, но другие методы могут возвращать их. Например, как мы видели в главе 10, в классе `URL` есть метод `openStream()`, возвращающий объект класса `InputStream`. Этот объект можно применять для получения

данных, переданных по сети. Как указывалось, классы `InputStream` и `OutputStream` позволяют читать и записывать только отдельные байты и массивы байтов. В них нет методов для чтения и записи строк и чисел. Для этого нужно создать специальные дочерние классы. Например, классы `DataInputStream` и `DataOutputStream` позволяют работать со всеми основными типами данных, предусмотренными в языке Java.

Для текста в кодировке Unicode используются классы, производные от классов `Reader` и `Writer`. Основные методы классов `Reader` и `Writer` аналогичны методам классов `InputStream` и `OutputStream`.

```
abstract int read()
abstract void write(int b)
```

Они работают так же, как и соответствующие методы из классов `InputStream` и `OutputStream`, за исключением, разумеется, того факта, что метод `read` возвращает либо символ в кодировке Unicode (в виде целого числа в диапазоне от 0 до 65535), либо число -1, если обнаружен конец файла.

Наконец, отметим, что существуют потоки, например `ZipInputStream` и `ZipOutputStream`, позволяющие читать и записывать файлы в формате ZIP.

В JDK 5.0 были предложены четыре новых интерфейса: `Closeable`, `Flushable`, `Readable` и `Appendable` (рис. 12.3). Первые два интерфейса очень просты. В каждом из них объявлен один из приведенных ниже методов.

```
void close() throws IOException
void flush()
```

Классы `InputStream`, `OutputStream`, `Reader` и `Writer` реализуют интерфейс `Closeable`, а `OutputStream` и `Writer` — интерфейс `Flushable`

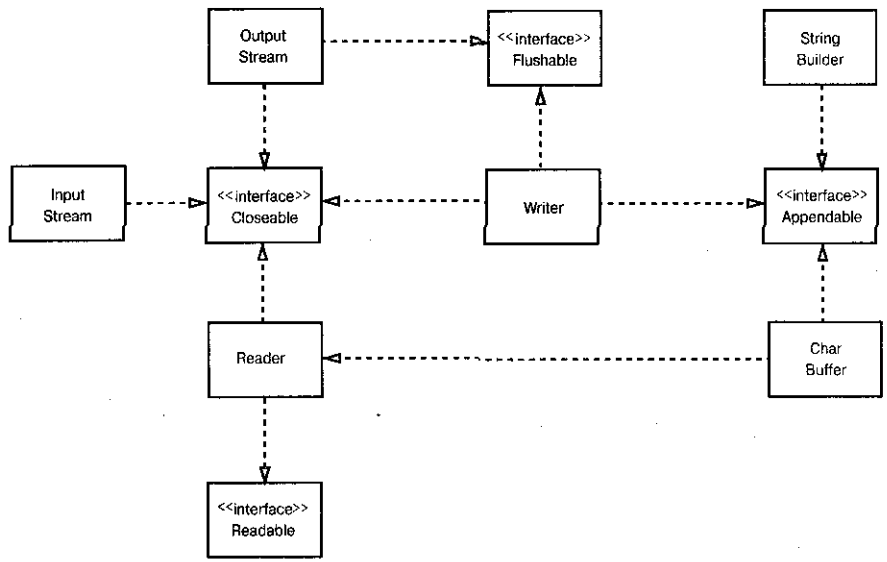


Рис. 12.3. Интерфейсы `Closeable`, `Flushable`, `Readable` и `Appendable`

В интерфейсе Readable объявлен единственный метод

```
int read(CharBuffer cb)
```

Класс CharBuffer содержит методы для чтения и записи с последовательным или произвольным доступом.

В интерфейсе Appendable объявлены два метода, предназначенные для записи в конец файла отдельных символов и символьных последовательностей.

```
Appendable append(char c)
Appendable append(CharSequence s)
```

Тип CharSequence представлен интерфейсом, описывающим минимальный набор свойств последовательности символов. Этот интерфейс реализуют классы String, CharBuffer и StringBuilder/StringBuffer.

Из всего семейства классов, ориентированных на работу с потоками, только Writer реализует интерфейс Appendable.

java.io.Closeable 5.0

- `void close()`

  Закрывает поток, представленный объектом Closeable. Данный метод может генерировать исключение IOException.

java.io.Flushable 5.0

- `void flush()`

  Принудительно выводит данные из буфера объекта Flushable.

java.lang.Readable 5.0

- `int read(CharBuffer cb)`

  Предпринимает попытку прочитать в cb столько символов, сколько он может хранить. Возвращает число прочитанных символов или значение -1, если данные из объекта Readable недоступны.

java.lang.Appendable 5.0

- `Appendable append(char c)`

  Добавляет коды, представляющие символ, к объекту Appendable. Возвращает ссылку this.

- `Appendable append(CharSequence cs)`

  Добавляет к объекту Appendable все коды, содержащиеся в cs. Возвращает ссылку this.

java.lang.CharSequence 1.4

- `char charAt(int index)`

  Возвращает код символа с указанным индексом.

- `int length()`

  Возвращает число кодовых единиц, представляющих символы, которые содержатся в последовательности.

- `CharSequence subSequence(int startIndex, int endIndex)`

  Возвращает объект `CharSequence`, содержащий коды, которые в исходной последовательности находятся в позициях от `startIndex` до `endIndex - 1`.

- `String toString()`

  Возвращает строку, состоящую из символов последовательности.

Фильтры потоков

Классы `FileInputStream` и `FileOutputStream` представляют потоки, связанные с файлами на диске. Имя файла или полный путь к нему задается в конструкторе. Например, с помощью следующего выражения можно связать поток с файлом `"employee.dat"`, находящемся в текущем каталоге:

```
FileInputStream fin = new FileInputStream("employee.dat");
```

Поскольку обратная косая черта (\)является управляющим символом, при записи путей в среде Windows следует использовать последовательность `"\\"` (например `"C:\\Windows\\win.ini"`). Кроме того, в среде Windows можно использовать обычную косую черту (например, `"C:/Windows/win.ini"`), поскольку большинство файловых систем на платформе Windows интерпретирует косую черту как разделитель файлов. Однако пользоваться этим разделителем не рекомендуется, поскольку поведение функций в системе Windows постоянно меняется, а в других операционных системах могут использоваться другие разделители файлов. Для того чтобы гарантировать независимость программы от конкретной платформы, нужно применять правильный разделитель файлов. Он хранится в строке `File.separator`.

В конструкторе `FileInputStream` можно также использовать объекты класса `File` (они подробно описываются в конце главы).

```
File f = new File("employee.dat");
FileInputStream fin = new FileInputStream(f);
```

Подобно `InputStream` и `OutputStream`, классы `FileInputStream` и `FileOutputStream` поддерживают чтение и запись только на уровне байтов. Таким образом, из объекта `fin` можно читать лишь отдельные байты или массивы байтов.

```
byte b = (byte)fin.read();
```

Поскольку все классы из пакета `java.io` интерпретируют относительные имена путей, считая отправной точкой текущий каталог, может понадобиться определить имя данного каталога. Для этого следует вызвать метод `System.getProperty("user.dir")`.

Создав экземпляр класса `DataInputStream`, можно читать числовые данные.

```
dataInputStream din = ...;
double s = din.readDouble();
```

Однако подобно тому, как в классе `FileInputStream` нет методов для чтения числовых данных, класс `DataInputStream` не имеет методов для чтения информации из файла.

В языке Java предусмотрен продуманный механизм для разделения ответственности. Некоторые потоки (например, `FileInputStream` и поток ввода, возвращаемый методом `openStream()` класса `URL`) могут извлекать байты из файла и других более экзотических источников. Другие потоки (такие как `DataInputStream` и `PrintWriter`) могут объединять байты, создавая более полезные типы данных. Программист, работающий на языке Java, должен объединять эти две разновидности потоков, создавая так называемые *фильтрованные потоки*. Для этого нужно передать существующий поток конструктору другого потока. Например, для считывания чисел из файла необходимо сначала создать объект класса `FileInputStream`, а затем передать его конструктору класса `DataInputStream`.

```
FileInputStream fin = new FileInputStream("employee.dat");
DataInputStream din = new DataInputStream(fin);
double s = din.readDouble();
```

Следует иметь в виду, что поток, созданный с помощью приведенного выше кода, *по-прежнему имеет доступ* к данным, хранящимся в соответствующем файле. Однако теперь он приобрел новые функциональные возможности.

На рис. 12.1 вы видели классы `FilterInputStream` и `FilterOutputStream`. Объединяя их дочерние классы, можно создать свой собственный поток. Например, по умолчанию буферизация потоков не предусмотрена. Это значит, что при каждой операции чтения происходит обращение к операционной системе за очередным байтом. Чтобы организовать буферизацию при вводе данных из файла `employee.dat`, хранящегося в текущем каталоге, нужно использовать следующую последовательность вызовов конструкторов:

```
DataInputStream din = new DataInputStream(
   new BufferedInputStream(
      new FileInputStream("employee.dat")));
```

Обратите внимание на то, что конструктор `DataInputStream` в этой цепочке вызывается последним, поскольку мы хотим одновременно использовать его методы, которые обращались бы к методу `read()`, поддерживающему буферизованный ввод. Несмотря на то что приведенный выше код достаточно громоздкий, это неизбежно: *мы должны углублять иерархию конструкторов потоков, пока не достигнем нужного* результата.

Иногда возникает необходимость отслеживать состояние промежуточных потоков, связанных в цепочку. Например, при считывании данных часто требуется предварительно просмотреть следующий байт и определить, хранится ли там ожидаемое значение. Для этой цели в языке Java предусмотрен класс `PushbackInputStream`.

```
PushbackInputStream pbin = new PushbackInputStream(
   new BufferedInputStream(
      new FileInputStream("employee.dat")));
```

Теперь можно прочитать очередной байт и вернуть его обратно, если его значение нам не подходит.

```
int b = pbin.read();
if (b != '<') pbin.unread(b);
```

Однако чтение и возврат данных можно выполнять только с потоками класса PushbackInputStream. Если нужно иметь возможность просматривать заранее и возвращать числа, следует одновременно применить потоки PushbackInputStream и DataInputStream.

```
DataInputStream din = new DataInputStream(
   pbin = new PushbackInputStream(
      new BufferedInputStream(
         new FileInputStream("employee.dat"))));
```

В библиотеках потоков на других языках такие операции, как буферизация и предварительный просмотр, предусмотрены автоматически, поэтому необходимость образовывать иерархические цепочки конструкторов, работая на языке Java, создает некоторые трудности. И все же возможность подбирать фильтры для создания нужной последовательности потоков обеспечивает невероятную гибкость. Например, можно считывать числа из архивных ZIP-файлов. Для этого нужно создать следующую последовательность потоков (рис. 12.4):

```
ZipInputStream zin
   = new ZipInputStream(new FileInputStream("employee.zip"));
DataInputStream din = new DataInputStream(zin);
```

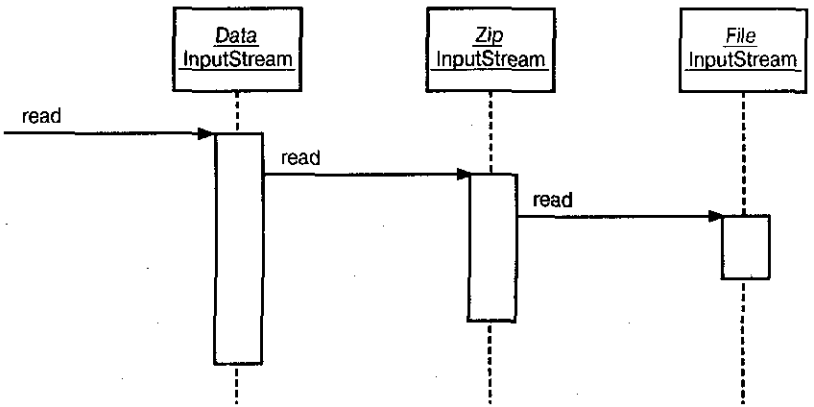


Рис. 12.4. Последовательность профильтрованных потоков

(Потоки, связываемые с ZIP-файлами, будут рассматриваться далее в этой главе.)

Если не принимать во внимание довольно громоздкие конструкции, необходимые для фильтрации потоков, возможность подбирать нужные потоки представляет собой весьма полезное свойство языка Java.

API **java.io.FileInputStream 1.0**

- `FileInputStream(String name)`

  Создает новый поток ввода, связанный с файлом, имя которого задается строкой name.

- `FileInputStream(File f)`

  Создает новый поток ввода, связанный с файлом. При создании потока используется информация, инкапсулированная в объекте `File`.

java.io.FileOutputSt

- `FileOutputStream(String name)`

  Создает новый поток вывода, связанный с файлом, имя которого задается строкой `name`. Путь к файлу считается относительным и отсчитывается от текущего каталога. Обратите внимание на то, что этот метод автоматически удаляет *любой ранее существовавший файл с указанным именем.*

- `FileOutputStream(String name, boolean append)`

  Создает новый поток вывода, связанный с файлом, имя которого задается строкой `name`. Путь к файлу считается относительным и отсчитывается от текущего каталога. Если параметр `append` имеет значение `true`, данные дописываются в конец файла. В этом случае существующий файл с указанным именем не удаляется.

- `FileOutputStream(File f)`

  Создает новый поток вывода, связанный с файлом. При создании потока используется информация, инкапсулированная в объекте `File`. Обратите внимание на то, что этот метод автоматически удаляет любой ранее существовавший файл, имя которого совпадает с именем файла `f`.

java.io.BufferedInputStream 1.0

- `BufferedInputStream(InputStream in)`

  Создает новый буферизованный поток ввода. Размер буфера задается по умолчанию. При работе с буферизованным потоком нет необходимости обращаться к устройству при выполнении каждой операции чтения. Если буфер пуст, в него помещается новый блок данных.

- `BufferedInputStream(InputStream in, int n)`

  Создает новый буферизованный поток ввода. Размер буфера задается пользователем.

java.io.BufferedOutputStream 1.0

- `BufferedOutputStream(OutputStream out)`

  Создает новый буферизованный поток вывода. Размер буфера задается по умолчанию. При использовании данного потока нет необходимости обращаться к устройству при каждой операции вывода. Если буфер полностью заполнен, данные выводятся. Также поддерживается операция принудительного вывода.

- `BufferedOutputStream(OutputStream out, int n)`

  Создает новый буферизованный поток вывода. Размер буфера задается пользователем.

API java.io.PushbackInputStream 1.0

- `PushbackInputStream(inputStream in)`

  Создает поток, в котором можно предварительно просмотреть следующий байт.

- `PushbackInputStream(inputStream in, int size)`

  Создает поток с буфером заданного размера, в котором можно предварительно просмотреть следующий байт.

- `void unread(int b)`

  "Заталкивает" прочитанный байт обратно. Этот байт будет вновь извлечен при выполнении следующей операции чтения.

  Параметр `b` Байт, который будет прочитан повторно

Потоки данных

В процессе работы часто возникает необходимость записать результат вычислений или прочитать его. В потоках предусмотрены методы для повторного чтения данных всех основных типов языка Java. Для того чтобы записать число, символ, логическое значение или строку, используется один из перечисленных ниже методов интерфейса `DateOutput`.

```
writeChars()
writeByte()
writeInt()
writeShort()
writeLong()
writeFloat()
writeDouble()
writeChar()
writeBoolean()
writeUTF()
```

Например, метод `writeInt()` всегда записывает целое число, независимо от количества его цифр, в 4-байтовом двоичном представлении, а метод `writeDouble()` всегда записывает число типа `double`, представленное в 8-байтовом виде. Содержимое файла, полученное в результате подобных операций вывода, не подходит для восприятия человеком. Однако при этом любое значение определенно занимает одинаковый объем памяти. Это значительно ускоряет процесс чтения данных. (Как вывести числа в удобном для пользователя виде, вы узнаете в разделе, посвященном классу `PrintWriter`.)

В зависимости от платформы числа в памяти могут храниться двумя способами. Допустим, что вы работаете с 4-байтовыми величинами, скажем, числами типа `int`. Их можно записать таким образом, чтобы первый байт в памяти содержал старшие разряды числа (порядок записи big-endian), а можно записать в них младшие разряды (порядок записи little-endian). Например, на платформе SPARC используется порядок big-endian, а на платформе Pentium — little-endian. Это может порождать проблемы. Например, при записи файла из программы на языке C или C++ данные будут сохранены в том порядке, в каком они хранятся в процессоре, и при переносе такой программы с одной платформы на другую возникнут недоразумения. В языке Java все числа записываются в порядке big-endian. Это гарантирует независимость от платформы файлов, созданных программами на языке Java.

Метод `writeUTF()` записывает строковые данные с использованием модифицированной версии Unicode Text Format (UTF). Вместо того чтобы применить стандартную кодировку UTF-8 (табл. 12.1), строки сначала представляются в UTF-16 (табл. 12.2), а затем результат кодируется посредством правил UTF-8. Для символов с кодами, превышающими 0xFFFF, применяются специальные правила. Такой подход используется для обеспечения обратной совместимости с виртуальными машинами, разработанными тогда, когда длина Unicode-символов была ограничена 16 битами.

Поскольку данная модификация UTF-8 не используется в других языках, метод `writeUTF()` следует применять для записи только тех строк, которые предназначены для обработки виртуальной машиной Java. Такой подход можно использовать в программе, генерирующей байтовый код. Для других целей следует применять метод `writeChars()`.

Таблица 12.1. Кодировка UTF-8

Диапазон символов	Код
0...7F	$0a_6a_5a_4a_3a_2a_1a_0$
80...7FF	$110a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$
800...FFFF	$1110a_{15}a_{14}a_{13}a_{12}$ $10a_{11}a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$
10000...10FFFF	$11110a_{20}a_{19}a_{18}$ $10a_{17}a_{16}a_{15}a_{14}a_{13}a_{12}$ $10a_{11}a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$

Таблица 12.2. Кодировка UTF-16

Диапазон символов	Код
0...FFFF	$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8$ $a_7a_6a_5a_4a_3a_2a_1a_0$
10000...10FFFF	$110110b_{19}b_{18}$ $b_{17}a_{16}a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}$ $110111a_9a_8$ $a_7a_6a_5a_4a_3a_2a_1a_0$ где $b_{19}b_{18}b_{17}b_{16} = a_{20}a_{19}a_{18}a_{17}a_{16} - 1$

Описание форматов UTF-8 и UTF-16 можно найти в документах RFC 2279 (`http://ietf.org/rfc/rfc2279.txt`) и RFC 2781 (`http://ietf.org/rfc/rfc2781.txt`).

Для того чтобы прочитать данные вновь, используются следующие методы:

```
readInt()                readDouble()
readShort()              readChar()
readLong()               readBoolean()
readFloat()              readUTF()
```

Двоичный формат данных компактен и не зависит от платформы, на которой выполняется программа. За исключением строк UTF, он удобен для произвольного доступа. Его основной недостаток — плохая читаемость.

`java.io.DataInput 1.0`

- `boolean readBoolean()`
  Читает логическое значение.

- `byte readByte()`

  Читает байт, состоящий из 8 битов.

- `char readChar()`

  Читает 16-битовый символ в кодировке Unicode.

- `double readDouble()`

  Читает 64-битовое число типа `double`.

- `float readFloat()`

  Читает 32-битовое число типа `float`.

- `void readFully(byte[] b)`

  Читает байты в массив `b`. Выполнение программы блокируется, пока массив не будет заполнен.

  Параметр `b` Буфер, в который помещаются данные

- `void readFully(byte[] b, int off, int len)`

  Читает байты в указанную часть массива `b`. Выполнение программы блокируется, пока фрагмент массива не будет заполнен.

  Параметры `b` Буфер, в который помещаются данные
  `off` Начальное смещение данных
  `len` Максимальное количество считываемых байтов

- `int readInt()`

  Читает 32-битовое целое число.

- `String readLine()`

  Читает строку, заканчивающуюся управляющими символами `"\n"`, `"\r"`, `"\r\n"` или маркером конца файла `EOF`. Возвращаемая строка содержит символы исходной строки, преобразованные в формат Unicode.

- `long readLong()`

  Читает 64-битовое целое число `Long`.

- `short readShort()`

  Читает 16-битовое целое число `Short`.

- `String readUTF()`

  Читает строку символов в формате UTF.

- `int skipBytes(int n)`

  Пропускает `n` байтов, блокируя выполнение программы, пока все байты не будут пропущены.

  Параметр `n` Количество пропускаемых байтов

`java.io.DataOutput 1.0`

- `void writeBoolean(boolean b)`

Записывает логическое значение.

- `void writeByte(byte b)`

  Записывает 8-битовый байт.

- `void writeChar(char c)`

  Записывает 16-битовый символ в кодировке Unicode.

- `void writeChars(String s)`

  Записывает все символы из строки.

- `void writeDouble(double d)`

  Записывает 64-битовое число `double`.

- `void writeFloat(float f)`

  Записывает 32-битовое число `float`.

- `void writeInt(int i)`

  Записывает 32-битовое целое число `int`.

- `void writeLong(long l)`

  Записывает 64-битовое целое число `Long`.

- `void writeShort(short s)`

  Записывает 16-битовое целое число `Short`.

- `void writeUTF(String s)`

  Записывает строку символов в формате UTF.

Потоки для работы с файлами в режиме произвольного доступа

Класс `RandomAccessFile` обеспечивает чтение и запись данных в любой позиции файла. Он одновременно реализует интерфейсы `DataInput` и `DataOutput`. Файлы, записанные на дисках, позволяют обращаться к ним в режиме произвольного доступа, а потоки данных, связанные с сетевыми соединениями, — нет. Файлы с произвольным доступом могут открываться либо только для чтения, либо для чтения и записи. Это зависит от второго параметра конструктора, который принимает значение "`r`" (только чтение) или "`rw`" (чтение и запись).

```
RandomAccessFile in = new RandomAccessFile("employee.dat", "r");
RandomAccessFile out = new RandomAccessFile("employee.dat", "rw");
```

Если существующий файл открывается с помощью `RandomAccessFile`, он не удаляется.

Файл с произвольным доступом имеет *указатель файла*, который определяет положение очередной записи. Метод `seek()` устанавливает этот указатель на произвольный байт внутри файла. Параметр метода `seek()` имеет тип `long` и принимает значение от 0 до максимальной длины файла, выраженной в байтах.

Метод `getFilePointer()` возвращает текущее значение указателя файла.

Чтобы прочитать данные из файла с произвольным доступом, используются те же методы, что и для объектов класса `DataInputStream`, например `readInt()` и `readUTF()`. Это не случайно, поскольку данные методы объявлены в интерфейсе

DataInput, который реализуется как классом DataInputStream, так и классом RandomAccessFile.

Аналогично для того, чтобы записать данные в файл с произвольным доступом, применяются те же методы writeInt() и writeUTF(), что и в классе DataOutputStream. Эти методы определены в интерфейсе DataOutput, общем для обоих классов.

Преимущество, получаемое в результате одновременной реализации классом RandomAccessFile интерфейсов DataInput и DataOutput, состоит в том, что программист может создавать методы, типы параметров которого могут определяться *интерфейсами* DataInput и DataOutput.

```
class Employee
{
   ...
   read(DataInput in) { ... }
   write(DataOutput out) { ... }
}
```

Обратите внимание на то, что метод read() может поддерживать объекты DataInputStream и RandomAccessFile, поскольку оба эти класса реализуют интерфейс DataInput. То же относится и к методу write().

java.io.RandomAccessFile 1.0

- `RandomAccessFile(String file, String mode)`
- `RandomAccessFile(File file, String mode)`

  Параметры *name* — *Файл, предназначенный для открытия*

  mode — Принимает значение "r", если файл открывается только для чтения, и "rw", если он открывается для чтения и записи

- `long getFilePointer()`

  Возвращает текущее значение указателя файла.

- `void seek(long pos)`

  Перемещает указатель файла в точку, отстоящую на pos байтов от начала файла.

- `long length()`

  Возвращает длину файла, выраженную в байтах.

Текстовые потоки

В предыдущем разделе мы обсудили ввод и вывод двоичных данных. Насколько двоичный ввод-вывод быстр и эффективен, настолько же он не удобен для восприятия человеком. Сейчас мы рассмотрим ввод-вывод *текста*. Например, если сохранить целое число 12234 в двоичном виде, оно будет выглядеть как последовательность байтов 00 00 04 D2 (в шестнадцатеричной системе). В текстовом формате это число будет сохранено в виде строки "1234".

К сожалению, при составлении Java-программы для этого нужно выполнить довольно много работы, поскольку в данном языке используется кодировка Unicode. Кодировка символа для строки "1234" фактически будет выглядеть как 00 31 00 32

00 33 00 34 (в шестнадцатеричной системе). Однако в настоящее время большинство сред, в которых выполняются программы на языке Java, используют свою собственную систему кодирования. В ней символы могут представляться одним, двумя или несколькими байтами. Например, в среде Windows строка записывается с помощью ASCII-кода как 31 32 33 34 без дополнительных нулевых байтов. Если текст в файле записан в формате Unicode, весьма вероятно, что получатель не сможет его прочитать. Чтобы преодолеть эту трудность, в языке Java предусмотрен набор фильтров, позволяющих преобразовать текст в формате Unicode в текст, записанный с помощью системы кодирования, принятой в локальной операционной системе. Все эти фильтры расширяют абстрактные классы `Reader` и `Writer`, а их имена напоминают имена классов, используемых для чтения и записи двоичных данных. Например, класс `InputStreamReader` преобразует поток ввода, содержащий байты, которые соответствуют определенной системе кодирования, в объект класса `Reader`, порождающий символы в системе кодирования Unicode. Аналогично класс `OutputStreamWriter` преобразует поток символов в системе Unicode в поток байтов, соответствующих заданной системе кодирования.

Рассмотрим, например, как создать объект для распознавания нажатых клавиш и кодировки соответствующих символов в формате Unicode.

```
InputStreamReader in = new InputStreamReader(Syaten.in)
```

Предполагается, что в локальной системе используется обычная система кодирования символов. Например, в операционной системе Windows применяется кодировка ISO 8859-1 (известная также под названием ISO Latin-1; программисты ее часто называют ANSI-код). В конструкторе класса `InputStreamReader` можно задать и другую систему кодирования.

```
InputStreamReader(InputStream, String)
```

Здесь строка, задаваемая в качестве второго параметра, описывает схему кодирования, которую вы хотите использовать.

```
InputStreamReader in = new InputStreamReader(
   new FileInputStream("kremlin.dat"), "ISO8859_5");
```

Наборы символов будут более подробно рассмотрены в следующем разделе.

Поскольку вполне естественно связать объекты для чтения и записи данных с файлом, была создана пара классов: `FileReader` и `FileWriter`. Например, определение объекта

```
FileWriter out = new FleWriter("output.txt");
```

эквивалентно определению

```
OutputStreamWriter out = new OutputStreamWriter(new
   FileOutputStream("output.txt"));
```

Наборы символов

В прошлом поддержка наборов символов, предназначенных для отображения текстов на различных языках, носила скорее эпизодический характер. Пакет `java.nio` и класс `Charset`, реализованные в JDK 1.4, положили начало унификации средств обработки наборов символов.

Набор символов отображает последовательности двухбайтовых кодов Unicode в байтовые последовательности, соответствующие локальным кодировкам. Одной из наиболее популярных кодировок является ISO-8859-1, в которой используется однобайтовое представление первых 256 символов Unicode. Все более популярной становится ISO-8859-15, в которой некоторые редко используемые символы ISO-8859-1 заменены буквами французского и финского языков, и, что более важно, символ ¤ *с кодом 0x44 заменен символом, представляющим денежную единицу "евро"*. В качестве примеров кодировок можно привести формат представления знаков переменным числом байтов, используемый в японском и китайском языках.

Класс `Charset` использует имена наборов символов, стандартизованные в IANA Character Set Registry (`http://www.iana.org/assignments/character-sets`). Эти имена слегка отличаются от имен, поддерживаемых предыдущими версиями JDK. Например, "официальное" имя ISO-8859-1 изменено с `ISO8859_1`, принятого в JDK 1.3, на `ISO-8859-1`. Чтобы обеспечить соответствие другим соглашениям об именовании, для каждого набора символов установлен ряд псевдонимов. Например, ISO-8859-1 имеет следующие псевдонимы:

```
ISO8859-1
ISO_8859_1
ISO8859_1
ISO_8859-1
ISO_8859-1:1987
8859_1
latin1
l1
csISOLatin1
iso-ir-100
cp819
IBM819
IBM-819
819
```

Имена наборов символов не зависят от регистра.

Для того чтобы создать экземпляр класса `Charset`, надо вызвать статический метод `forName()`, передав ему либо официальное имя набора, либо один из псевдонимов.

```
Charset cset = Charset.forName("ISO-8859-1");
```

Метод `aliases()` возвращает объект `Set`, содержащий псевдонимы. Тип `Set` будет рассмотрен во втором томе, здесь же мы приведем лишь фрагмент кода, реализующий перебор элементов набора.

```
Set<String> aliases = cset.aliases();
for (String alias : aliases)
   System.out.println(alias);
```

Подробную информацию о наборе ISO 8859 можно найти по адресу `http://czyborra.com/charsets/iso8859.html`.

Международные версии Java поддерживают гораздо большее количество кодировок. Существует даже механизм для добавления наборов символов; подробно этот вопрос освещен в документации по JDK. Чтобы определить, какие наборы символов доступны в конкретной реализации, надо вызвать метод `availableCharsets()`,

который возвращает объект SortedMap. Приведенный ниже фрагмент кода позволяет определить имена доступных наборов.

```
Set<String, Charset> charsets = Charset.availableCharsets();
for (String name : charsets.keySet())
   System.out.println(name);
```

В табл. 12.3 перечислены кодировки, которые должна поддерживать каждая реализация Java. В табл. 12.4 приведены системы кодирования, инсталлируемые по умолчанию. Наборы, информация о которых содержится в табл. 12.5 и 12.6, устанавливаются только в тех операционных системах, в которых применяются языки неевропейских стран. Кодировки, описанные в табл. 12.6, предназначены для обеспечения совместимости с предыдущими версиями JDK.

В локальных кодировках не могут быть представлены все символы Unicode. Символы, которые невозможно воспроизвести, заменяются знаком ?.

Таблица 12.3. Обязательные наборы символов

Стандартное имя набора символов	Традиционно используемое имя	Описание
US-ASCII	ASCII	Американский стандартный код для обмена информацией
ISO-8859-1	ISO8859_1	ISO 8859-1, латинский алфавит №1
UTF-8	UTF8	Восьмибитовый формат UTF (Unicode Transformation Format)
UTF-16	UTF-16	Шестнадцатибитовый формат UTF. Порядок следования байтов определяется необязательным параметром
UTF-16BE	UnicodeBigUnmarked	Шестнадцатибитовый формат UTF, порядок следования байтов big-endian
UTF-16LE	UnicodeLittleUnmarked	Шестнадцатибитовый формат UTF, порядок следования байтов little-endian

Таблица 12.4. Базовые наборы символов

Стандартное имя набора символов	Традиционно используемое имя	Описание
ISO8859-2	ISO8859_2	ISO 8859-2, латинский алфавит №2
ISO8859-4	ISO8859_4	ISO 8859-4, латинский алфавит № 4
ISO8859-5	ISO8859_5	ISO 8859-5, латинский алфавит/кириллица
ISO8859-7	ISO8859_7	ISO 8859-7, латинский/греческий алфавит
ISO8859-9	ISO8859_9	ISO 8859-9, латинский алфавит № 5
ISO8859-13	ISO8859_13	ISO 8859-13, латинский алфавит № 7
ISO8859-15	ISO8859_15	ISO 8859-15, латинский алфавит № 9
windows-1250	Cp1250	Восточноевропейский алфавит для Windows
windows-1251	Cp1251	Кириллица для Windows
windows-1252	Cp1252	Латинский алфавит № 1 для Windows
windows-1253	Cp1253	Греческий алфавит для Windows
windows-1254	Cp1254	Турецкий алфавит для Windows
windows-1257	Cp1257	Алфавит стран Балтии для Windows

```
                getMessage(),
                "Title",
                getType(messageTypePanel));
          else if (typePanel.getSelection().equals("Option"))
             JOptionPane.showOptionDialog(
                OptionDialogFrame.this,
                getMessage(),
                "Title",
                getType(optionTypePanel),
                getType(messageTypePanel),
                null,
                getOptions(),
                getOptions()[0]);
       }
    }

    public static final int DEFAULT_WIDTH = 600;
    public static final int DEFAULT_HEIGHT = 400;

    private ButtonPanel typePanel;
    private ButtonPanel messagePanel;
    private ButtonPanel messageTypePanel;
    private ButtonPanel optionTypePanel;
    private ButtonPanel optionsPanel;
    private ButtonPanel inputPanel;

    private String messageString = "Message";
    private Icon messageIcon = new ImageIcon("blue-ball.gif");
    private Object messageObject = new Date();
    private Component messageComponent = new SamplePanel();
 }

/**
   Панель с графическим изображением.
*/

class SamplePanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;
      Rectangle2D rect = new Rectangle2D.Double(0, 0,
         getWidth() - 1, getHeight() - 1);
      g2.setPaint(Color.YELLOW);
      g2.fill(rect);
      g2.setPaint(Color.BLUE);
      g2.draw(rect);
   }

   public Dimension getMinimumSize()
   {
      return new Dimension(10, 10);
   }
}
```

Продолжение табл. 12.6

Имя	Название
Cp284	IBM, каталанский/испанский, испанский для стран Латинской Америки
Cp285	IBM, Соединенное королевство, Ирландия
Cp297	IBM, Франция
Cp420	IBM, арабский
Cp424	IBM, иврит
Cp437	MS-DOS, США, Австралия, Новая Зеландия, Южная Африка
Cp500	EBCDIC 500V1
Cp737	PC, Греция
Cp775	PC, страны Балтии
Cp838	IBM, Тайланд, расширение SBCS
Cp850	MS-DOS, латинский алфавит № 1
Cp852	MS-DOS, латинский алфавит № 2
Cp855	IBM, кириллица
Cp856	IBM, иврит
Cp857	IBM, Турция
Cp858	Вариант Cp850 с символом для представления евро
Cp860	MS-DOS, Португалия
Cp861	MS-DOS, Исландия
Cp862	PC, иврит
Cp863	MS-DOS, Канада, французский
Cp864	PC, арабский
Cp865	MS-DOS, скандинавские страны
Cp866	MS-DOS, Россия
Cp868	MS-DOS, Пакистан
Cp869	IBM, современный греческий
Cp870	IBM, латинский алфавит № 2 с многоязыковым расширением
Cp871	IBM, Исландия
Cp874	IBM, Тайланд
Cp875	IBM, Греция
Cp918	IBM, Пакистан (Urdu)
Cp921	IBM Латвия, Литва (AIX, DOS)
Cp922	IBM Эстония (AIX, DOS)
Cp930	Япония, Katakana-Kanji и 4370 UDC, расширение 5026

Продолжение табл. 12.6

Имя	Название
Cp933	Корея и 1880 UDC, расширение 5029
Cp935	Упрощенный набор китайских символов и 1880 UDC, расширение 5031
Cp937	Традиционный набор китайских символов и 6204 UDC, расширение 5033
Cp939	Япония, Latin Kanji и 4370 UDC, расширение 5035
Cp942	IBM 0S/2 Япония, расширение Cp932
Cp942C	Вариант Cp942
Cp943	IBM OS/2, Япония, расширение Cp932 и Shift-JIS
Cp943C	Вариант Cp943
Cp948	OS/2, Китай (Тайвань), расширение 938
Cp949	PC, Корея
Cp949C	Вариант Cp949
Cp950	PC, Китай (Гонконг, Тайвань)
Cp964	AIX, Китай (Тайвань)
Cp970	AIX, Корея
Cp1006	IBM AIX, Пакистан (Urdu)
Cp1025	IBM, кириллица с многоязыковой поддержкой: Болгария, Босния, Герцеговина, Македония (FYR)
Cp1026	IBM, латинский алфавит № 5, Турция
Cp1046	IBM, арабский для Windows
Cp1097	IBM, Иран (фарси)/персидский
Cp1098	IBM, Иран (фарси)/персидский (PC)
Cp1112	IBM, Латвия, Литва
Cp1122	IBM, Эстония
Cp1123	IBM, Украина
Cp1124	IBM AIX, Украина
Cp1140	Вариант Cp037 с символом для представления евро
Cp1141	Вариант Cp273 с символом для представления евро
Cp1142	Вариант Cp277 с символом для представления евро
Cp1143	Вариант Cp278 с символом для представления евро
Cp1144	Вариант Cp280 с символом для представления евро
Cp1145	Вариант Cp284 с символом для представления евро
Cp1146	Вариант Cp285 с символом для представления евро
Cp1147	Вариант Cp297 с символом для представления евро
Cp1148	Вариант Cp500 с символом для представления евро

Окончание табл. 12.6

Имя	Название
Cp1149	Вариант Cp871 с символом для представления евро
Cp1381	IBM OS/2, DOS, Китай
Cp1383	IBM AIX, Китай
Cp33722	IBM-eucJP, Япония (расширение 5050)
ISO2022CN	ISO 2022 CN, Китай (преобразование только в Unicode)
ISO2022CN_CNS	CNS 11643, традиционный набор китайских символов в форме ISO 2022 CN (преобразование только из Unicode)
ISO2022CN_GB	GB 2312, упрощенный набор китайских символов в форме ISO 2022 CN (преобразование только из Unicode)
JIS0201	JIS X 0201, Япония
JIS0208	JIS X 0208, Япония
JIS0212	JIS X 0212, Япония
JISAutoDetect	Распознавание и преобразование Shift-JIS, EUC-JP, ISO 2022 JP (преобразование только в Unicode)
MS874	Windows, Тайланд
MacArabic	Macintosh, арабский
MacCentralEurope	Macintosh, латинский алфавит № 2
MacCirillic	Macintosh, кириллица
MacGreek	Macintosh, Греция
MacHebrew	Macintosh, иврит
MacIceland	Macintosh, Исландия
MacRomania	Macintosh, Румыния
MacThai	Macintosh, Тайланд
MacTurkish	Macintosh, Турция
MacUkraine	Macintosh, Украина

Имея набор символов, вы можете использовать его для преобразования строк Unicode в кодированные байтовые последовательности. Пример подобного преобразования приведен ниже.

```
String str = . . .;
ByteBuffer buffer = cset.encode(str);
byte[] bytes = buffer.array();
```

Для обратного преобразования необходим байтовый буфер. Преобразовать обычный байтовый массив в байтовый буфер позволяет статический метод `wrap()`, определенный в классе `ByteBuffer`. Результатом выполнения метода `decode()` является объект `CharBuffer`. Преобразовать его в строку позволяет метод `toString()`.

```
byte[] bytes = . . .;
ByteBuffer bbuf = ByteBuffer.wrap(bytes, offset, length);
CharBuffer cbuf = cset.decode(bbuf);
String str = cbuf.toString();
```

java.nio.charset.Charset 1.4

- `static SortedMap availableCharsets()`

  Предоставляет информацию о доступных наборах символов для данной виртуальной машины. Возвращает объект `SortedMap`, в котором ключами являются имена наборов символов, а значения — наборами.

- `static Charset forName(String name)`

  Возвращает набор символов, соответствующий указанному имени.

- `Set aliases()`

  Возвращает набор псевдонимов для набора символов.

- `ByteBuffer encode(String str)`

  Преобразует указанную строку в байтовую последовательность.

- `CharBuffer decode(ByteBuffer buffer)`

  Декодирует последовательность символов. Символы, которые невозможно распознать, преобразуются в знак замещения Unicode ('\uFFFD').

java.nio.ByteBuffer 1.4

- `byte[] array()`

  Возвращает массив байтов, соответствующий буферу.

- `static ByteBuffer wrap(byte[] bytes)`
- `static ByteBuffer wrap(byte[] bytes, int offset, int length)`

  Возвращает буфер, соответствующий байтовому массиву или его части.

java.nio.CharBuffer

- `char[] array()`

  Возвращает массив кодовых единиц, управляемый буфером.

- `char charAt(int index)`

  Возвращает кодовую единицу с указанным индексом.

- `String toString()`

  Возвращает строку, состоящую из кодовых единиц, управляемых буфером.

Запись текста в поток вывода

Для вывода текста следует использовать класс `PrinterWriter`. Объекты этого класса позволяют выводить строки и числа в текстовом формате. Подобно классу `DataOutputStream`, имеющему методы вывода, но не способному обращаться к устройству, класс `PrintWriter` нужно объединять с другим объектом.

```
PrintWriter out = new PrintWriter(new
   FileWriter("employee.txt"));
```

Объект `PrintWriter` можно также объединять с потоком вывода.

```
PrintWriter out = new PrintWriter(new
   FileOutputStream("employee.txt"));
```

Конструктор `PrintWriter(OutputStream)` автоматически добавляет объект класса `OutputStreamWriter`, предназначенный для преобразования Unicode-символов в байты потока.

Записать данные в объект класса `PrintWriter` можно с помощью методов `print()` и `println()`, которые обычно используются вместе с потоком `System.out`. Эти методы применяются для вывода чисел, имеющих тип `int`, `short`, `long`, `float` или `double`, символов, логических значений, строк и объектов.

В качестве примера рассмотрим следующий фрагмент программы:

```
String name = "Harry Hacker";
double salary = 75000;
out.ptint(name);
out.print(' ');
out.println(salary);
```

В результате выполнения этого фрагмента строка

```
Harry Hacker 75000
```

будет записана в поток `out`. Затем символы будут преобразованы в байты и записаны в файл `employee.txt`.

Ветераны программирования на языке Java могут спросить, что случилось с классами `PrintStream` и `System.out`? В версии Java 1.0 класс `PrintStream` просто усекал все символы, закодированные в системе Unicode, отбрасывая старшие байты, и получал ASCII-коды. И наоборот, метод `readLine()` класса `DataInputStream` преобразовывал ASCII-код символа в код системы Unicode, полагая, что старшие байты равны 0. Очевидно, что такой подход не обеспечивает независимость от платформы и удобочитаемость программ. В следующей версии (Java 1.1) этот недостаток был исправлен путем введения классов `Reader` и `Writer`. Для поддержки совместимости с существующими программами `System.in`, `System.out` и `System.err` остались потоками. Однако теперь классы `PrintStream` и `PrintWriter` конвертируют Unicode-символы, используя кодировку, предусмотренную по умолчанию в локальной операционной системе. Объекты типа `PrintStream` функционируют так же, как и объекты класса `PrintWriter`, в которых предусмотрены методы `print()` и `println()`, за одним исключением: они позволяют посылать им обычные байты данных с помощью методов `write(int)` и `write(byte[])`.

Метод `println` автоматически добавляет корректный символ конца строки, принятый в данной операционной системе (`"\r\n"` — Windows, `"\n"` — Unix и `"\r"` — Macintosh). Признак конца строки возвращает метод `System.getProperty("line.separator")`.

Если объект класса `Writer` задает *режим автоматического сброса буфера* (autoflash mode), то при вызове метода `println()` данные выводятся принудительно, т.е. все символы, находящиеся в буфере, будут отправлены по назначению. По умолчанию режим автоматического сброса *отключен*. Включить или выключить этот режим мож-

но с помощью конструктора PrintWriter(Writer, boolean), задав в качестве второго параметра соответствующее логическое значение.

```
PrintWriter out = new PrintWriter(new
  FileWriter("employee.txt"), true); // Автоматический сброс.
```

Методы print() не генерируют никаких исключений. Для проверки ошибок в потоке следует вызывать метод checkError().

Средствами PrintWriter нельзя передавать байты данных. Этот метод предназначен только для вывода текста.

java.io.PrintWriter 1.1

- `PrintWriter(Writer out)`

  Создает новый объект PrintWriter. Режим автоматического сброса буфера отключен.

  Параметр `out` Объект Writer, предназначенный для вывода символов

- `PrintWriter(Writer out, boolean autoFlush)`

  Создает новый объект PrintWriter.

  Параметры `out` Объект Writer, предназначенный для вывода символов

  `autoFlush` Если задано значение true, то при вызове метода println() содержимое буфера будет выводиться принудительно

- `PrintWriter(OutputStream out)`

  Создает новый объект PrintWriter. Режим автоматического сброса буфера отключен. Существующий объект OutputStream автоматически создает необходимый вспомогательный объект OutputStreamWriter.

  Параметр `out` Поток вывода

- `PrintWriter(OutputStream out, boolean autoFlush)`

  Создает новый объект PrintWriter на основе существующего объекта OutputStream. Режим автоматического сброса буфера можно включать или отключать.

  Параметры `out` Поток вывода

  `autoFlush` Если задано значение true, то при вызове метода println() содержимое буфера будет выводиться принудительно

- `void print(Object obj)`

  Выводит объект в виде строки, полученной в результате применения метода toString().

  Параметр `obj` Объект, подлежащий выводу

- `void print(String s)`

Выводит Unicode-строку.

- `void println(String s)`

  Выводит строку. Если установлен режим автоматического сброса, содержимое буфера выводится принудительно.

- `void print(char[] s)`

  Выводит массив Unicode-символов.

- `void print(char s)`

  Выводит Unicode-символ.

- `void print(int i)`

  Выводит целое число в текстовом формате.

- `void print(long i)`

  Выводит целое число типа `long` в текстовом формате.

- `void print(float f)`

  Выводит число типа `float` в текстовом формате.

- `void print(double d)`

  Выводит число двойной точности типа `double` в текстовом формате.

- `void print(boolean b)`

  Выводит логическое значение в текстовом формате.

- `boolean checkError()`

  Возвращает значение `true`, если обнаружена ошибка форматирования или вывода. После появления ошибки информация в потоке считается некорректной, и все вызовы метода `checkError()` будут возвращать значения `true`.

Чтение текста из потока ввода

Как мы знаем, для того, чтобы записать данные в двоичном виде, нужно использовать объект `DataOutputStream`, а для записи данных в текстовом формате применяется объект `PrintWriter`. Следовательно, логично было бы ожидать, что существует аналог класса `DataInputStream`, позволяющий читать данные в текстовом формате. Наиболее близким аналогом `DataInputStream` является класс `Scanner`, который мы активно используем в своей работе. Однако до появления JDK 5.0, единственным инструментом, пригодным для ввода текстовых данных, был класс `BufferedReader`. Он содержал метод `readLine()`, позволяющий читать текстовые строки. Экземпляры этого класса нужно объединять с источником входных данных.

```
BufferedReader in = new BufferedReader(new
   FileReader("employee.txt"));
```

Если дальнейший ввод данных невозможен, метод `readLine()` возвращает ссылку `null`. Типичный цикл для ввода данных выглядит следующим образом:

```
String line;
while ((line = in.readLine()) != null)
{
   Обработка строки
}
```

Класс `FileReader` конвертирует байты в символы системы Unicode. Для других источников входных данных нужно использовать класс `InputStreamReader`. В отличие от класса `PrintWriter` в классе `InputStreamReader` нет метода, позволяющего преобразовать байты в символы Unicode.

```
BufferedReader in2 = new BifferedReader(new
   InputStreamReader(System.in));
BufferedReader in3 = new BifferedReader(new
   InputStreamReader(url.openStream));
```

Для того чтобы считать числа, представленные в текстовом формате, нужно сначала прочитать строку, а затем преобразовать ее в число.

```
String s = in.readLine();
double x = Double.parseDouble(s);
```

Эти выражения рассчитаны на ввод чисел, записанных по одному в каждой строке. В противном случае нужно приложить больше усилий, например, разбить входную строку с помощью класса `StringTokenizer`. Пример, демонстрирующий данный подход, будет рассмотрен далее в этой главе.

В языке Java есть классы `StringReader` и `StringWriter`, позволяющие работать со строками так, как если бы они были потоками данных. Это довольно удобно, если один и тот же код нужно применить для обработки и строк, и данных, полученных из потока.

Потоки, связанные с ZIP-файлами

ZIP-файлы — это архивы, в которых один или несколько обычных файлов хранятся в сжатом виде. В языке Java 1.1 применяются форматы GZIP и ZIP. (Информацию об этих форматах можно найти в документах RFC 1950, RFC 1951 и RFC 1952 по адресу `http://www.faqs.org/rfcs`.) В этом разделе мы сосредоточим внимание на более известном формате ZIP, а с форматом GZIP читатель в случае необходимости разберется сам. (Этот формат мало отличается от формата ZIP.)

Классы для обработки ZIP-файлов находятся в пакете `java.util.zip`, а не в `java.io`, поэтому не забудьте добавить оператор `import`. Несмотря на то что классы GZIP и ZIP не относятся к пакету `java.io`, они являются подклассами классов `java.io.FilterInputStream` и `java.io.FilterOutputStream`. Пакеты `java.util.zip` также содержат классы для выявления ошибок с помощью избыточного кода (CRC).

Каждый ZIP-файл имеет заголовок, в котором хранится информация о нем, например имя файла и примененный метод сжатия. В языке Java для чтения ZIP-файлов используется класс `ZipInputStream`, при этом конструктор такого класса получает в качестве параметра объект `FileInputStream`. Затем нужно найти конкретный элемент архива. Для описания элементов используется класс `ZipEntry`, экземпляр которого возвращается методом `getNextEntry()`. Метод `read()` класса `ZipUnputStream` возвращает значение –1, если обнаружен конец текущего элемента (а не конец всего ZIP-архива). После этого следует вызвать метод `closeEntry` и прочитать следующий элемент. Ниже приведен фрагмент кода, выполняющий чтение ZIP-файла.

```
ZipInputStream zin = new ZipInputStream(new
   FileInputStream(zipname));
ZipEntry entry;
while ((entry = zin.getNextEntry()) != null)
{
   analyze entry;
    чтение содержимого zin;
   zin.closeEntry();
}
zin.close();
```

При чтении элемента ZIP-файла можно обойтись без низкоуровневого метода `read()`. Обычно программисты предпочитают использовать более сложные фильтры потоков. Например, для чтения текстового файла из ZIP-архива можно воспользоваться следующим циклом:

```
BufferedReader in = new BufferedReader(new
   InputStreamReader(zin));
String s;
while ((s = in.readLine()) != null)
   Обработка строки s;
```

Программа, приведенная в листинге 12.1, открывает ZIP-файл и отображает файлы, хранящиеся в архиве, в списке, расположенном в нижней части экрана. Если дважды щелкнуть на имени файла, в текстовой области отобразится его содержимое (рис. 12.5).

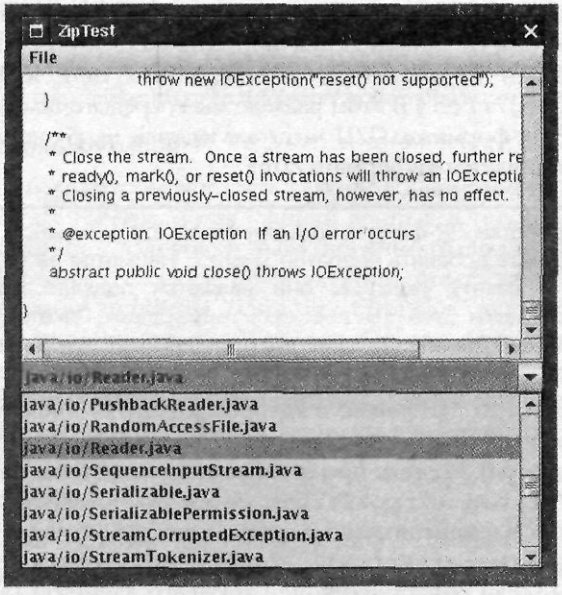


Рис. 12.5. Окно программы `ZipTest`

Листинг 12.1. Содержимое файла `ZipTest.java`

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import java.util.zip.*;
import javax.swing.*;
import javax.swing.filechooser.FileFilter;

public class ZipTest
{
   public static void main(String[] args)
   {
      ZipTestFrame frame = new ZipTestFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм с текстовой областью, предназначенной для отображения
   содержимого файла из ZIP-архива, раскрывающимся
   списком для выбора файлов в архиве и меню для
   загрузки нового архива.
*/
class ZipTestFrame extends JFrame
{
   public ZipTestFrame()
   {
      setTitle("ZipTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // Добавление строки меню и пунктов Open и Exit
      JMenuBar menuBar = new JMenuBar();
      JMenu menu = new JMenu("File");

      JMenuItem openItem = new JMenuItem("Open");
      menu.add(openItem);
      openItem.addActionListener(new OpenAction());

      JMenuItem exitItem = new JMenuItem("Exit");
      menu.add(exitItem);
      exitItem.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.exit(0);
            }
         });

      menuBar.add(menu);
      setJMenuBar(menuBar);

      // Добавление текстовой области и раскрывающегося списка
```

```
      fileText = new JTextArea();
      fileCombo = new JComboBox();
      fileCombo.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               loadZipFile((String) fileCombo.getSelectedItem());
            }
         });

      add(fileCombo, BorderLayout.SOUTH);
      add(new JScrollPane(fileText), BorderLayout.CENTER);
   }

   /**
      Обработчик для пункта меню File->Open.
   */
   private class OpenAction implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         // Окно для выбора файла архива.
         JFileChooser chooser = new JFileChooser();
         chooser.setCurrentDirectory(new File("."));
         ExtensionFileFilter filter = new ExtensionFileFilter();
         filter.addExtension(".zip");
         filter.addExtension(".jar");
         filter.setDescription("ZIP archives");
         chooser.setFileFilter(filter);
         int r = chooser.showOpenDialog(ZipTestFrame.this);
         if (r == JFileChooser.APPROVE_OPTION)
         {
            zipname = chooser.getSelectedFile().getPath();
            scanZipFile();
         }
      }
   }

   /**
      Просмотр содержимого zip-архива и заполнение списка.
   */
   public void scanZipFile()
   {
      fileCombo.removeAllItems();
      try
      {
         ZipInputStream zin = new ZipInputStream(new
            FileInputStream(zipname));
         ZipEntry entry;
         while ((entry = zin.getNextEntry()) != null)
         {
            fileCombo.addItem(entry.getName());
            zin.closeEntry();
         }
         zin.close();
```

```
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }

   /**
      Отображение файла из архива в текстовой области.
      @param name Имя файла в архиве
   */
   public void loadZipFile(String name)
   {
      try
      {
         ZipInputStream zin = new ZipInputStream(new
            FileInputStream(zipname));
         ZipEntry entry;
         fileText.setText("");

         // Поиск элемента, соответствующего имени.
         while ((entry = zin.getNextEntry()) != null)
         {
            if (entry.getName().equals(name))
            {
               // Чтение файла и отображение его содержимого
               // в текстовой области.
               BufferedReader in = new BufferedReader(new
                  InputStreamReader(zin));
               String line;
               while ((line = in.readLine()) != null)
               {
                  fileText.append(line);
                  fileText.append("\n");
               }
            }
            zin.closeEntry();
         }
         zin.close();
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }

   public static final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 300;

   private JComboBox fileCombo;
   private JTextArea fileText;
   private String zipname;
}

/**
   Данный фильтр выбирает все файлы с заданными
```

```
   расширениями. Использовался в программе FileChooserTest,
   которая рассматривалась в главе 9.
*/
class ExtensionFileFilter extends FileFilter
{
   /**
      Добавление расширения, которое должно распознаваться
      фильтром.
      @param extension Расширение (например, ".txt" или "txt")
   */
   public void addExtension(String extension)
   {
      if (!extension.startsWith("."))
         extension = "." + extension;
      extensions.add(extension.toLowerCase());
   }

   /**
      Задает описание файлов, распознаваемых фильтром.
      @param aDescription Описание распознаваемых файлов
   */
   public void setDescription(String aDescription)
   {
      description = aDescription;
   }

   /**
      Возвращает описание файлов,
      распознаваемых данным фильтром.
      @return Описание распознаваемых файлов
   */
   public String getDescription()
   {
      return description;
   }

   public boolean accept(File f)
   {
      if (f.isDirectory()) return true;
      String name = f.getName().toLowerCase();

      // Проверка расширения файла.
      for (String e : extensions)
         if (name.endsWith(e))
            return true;
      return false;
   }

   private String description = "";
   private ArrayList<String> extensions = new ArrayList<String>();
}
```

Если во время считывания ZIP-файла произошла ошибка, поток ввода ZIP-файлов генерирует исключение `ZipException`. Обычно эта ошибка возникает, если ZIP-файл поврежден.

Для того чтобы записать ZIP-файл, нужно открыть поток `ZipOutputStream`, передав его конструктору в качестве параметра объект `FileOutputStream`. Для каждого элемента, подлежащего сжатию, нужно создать объект `ZipEntry`. Конструктору `ZipEntry` передается имя файла. Другие параметры, например дата создания файла и метод распаковки, устанавливаются автоматически. Чтобы начать запись архива, нужно вызвать метод `putNextEntry()` класса `ZipOutputStream`. Затем файлы посылаются в поток ZIP. После этого вызывается метод `closeEntry()`. Эту процедуру необходимо повторять для каждого элемента, подлежащего архивации. Ниже приведена заготовка кода, реализующего описанный процесс.

```
FileOutputStream fout = new FileOutputStream("test.zip");
ZipOutputSream zout = new ZipOutputStream(fout);
для всех файлов
{
   ZipEntry ze = new ZipEntry(имя_файла);
   zout.putNextEntry(ze);
   передача данных в поток zout;
   zout.closeEntry();
}
zout.close();
```

JAR-файл представляет собой ZIP-архив, содержащий файл манифеста. Для чтения и записи манифеста используются классы `JarInputStream` и `JarOutputStream`.

Средства обмена с ZIP-архивами ярко иллюстрируют возможности потоков Java. И источник, и адресат ZIP-файлов могут быть чем угодно. Для того чтобы прочесть данные, хранящиеся в сжатой форме, нужно передать подходящий поток в конструктор ZIP-потока, причем соответствующий объект даже не узнает, что он считывает сжатые данные. Источником байтов в формате ZIP не всегда бывает файл — сжатые данные могут поступать из сети. Фактически JAR-файлы, которые обсуждались в главе 10, являются ZIP-файлами. Когда загрузчик классов некоторого аплета считывает JAR-файл, он читает и распаковывает данные, поступающие по сети.

Вопросы модификации ZIP-архивов подробно рассмотрены в статье, которую можно найти по адресу `http://www.javaworld.com/javaworld/jw-10-2000/jw-1027-toolbox.html`

`java.util.zip.ZipInputStream 1.1`

- `ZipInputStream(InputStream in)`

  Этот конструктор создает объект `ZipInputStream`, позволяющий загружать данные из потока `InputStream`.

  Параметр `in` Поток ввода

- `ZipEntry getNextEntry()`

  Возвращает объект класса `ZipEntry` для следующего элемента или ссылку `null`, если в архиве больше нет элементов.

- `void closeEntry()`

  Закрывает текущий открытый элемент в ZIP-файле. После этого с помощью метода `getNextEntry()` можно прочесть следующий элемент.

API `java.util.zip.ZipOutputStream`

- `ZipOutputStream(OutputStream out)`

  Этот конструктор создает объект `ZipOutStream`, позволяющий загружать данные из потока `OutputStream`.

  Параметр `out` Поток вывода

- `void putNextEntry(ZipEntry ze)`

  Записывает информацию из заданного объекта `ZipEntry` в поток и позиционирует указатель.

  Параметры `ze` Новый элемент

- `void closeEntry()`

  Закрывает текущий элемент ZIP-файла. Для перехода к следующему элементу используется метод `putNextEntry()`.

- `void setLevel(int level)`

  Устанавливает уровень сжатия, принимаемый по умолчанию для последующих элементов `DEFLATED`. Этот уровень обозначается константой `Deflater.DEFAULT_COMPRESSION`. Если уровень задан неправильно, генерируется исключение `IllegalArgumentException`.

  Параметр `level` Уровень сжатия в диапазоне от 0 (`NO_COMPRESSION`) до 9 (`BEST_COMPRESSION`)

- `void setMethod(int method)`

  Задает метод сжатия по умолчанию для класса `ZipOutputStream`. Этот метод используется для всех элементов, для которых метод не указан явно.

  Параметр `method` Метод сжатия (`DEFLATED` или `STORED`)

API `java.util.zip.ZipEntry`

- `ZipEntry(String name)`

  Параметр `name` Имя элемента

- `long getCrc()`

  Возвращает контрольную сумму CRC32 для объекта `ZipEntry`.

- `String getName()`

  Возвращает имя элемента.

- `long getSize()`

  Возвращает размер распакованного элемента или значение –1, если этот размер неизвестен.

- `boolean isDirectory()`

Возвращает признак того, является ли данный элемент каталогом.

- `void setMethod(int method)`

  Параметр `method` Метод сжатия для заданного элемента (`DEFLATED` или `STORED`)

- `void setSize(long size)`

  Задает размер элемента. Нужен только для метода `STORED`.

  Параметры `size` Размер распакованного элемента

- `void setCrc(long crc)`

  Устанавливает контрольную сумму CRC32 для элемента. Для вычисления этой суммы используется класс `CRC32`. Требуется только для метода `STORED`.

  Параметры `crc` Контрольная сумма элемента

`java.util.zip.ZipFile` 1.

- `ZipFile(String name)`

  Создает объект `ZipFile` для чтения.

  Параметры `name` Строка, содержащая путь к файлу

- `ZipFile(File file)`

  Создает объект класса `ZipFile` для чтения.

  Параметр `file` Файл для чтения

- `Enumeration entries()`

  Возвращает объект `Enumeration`, в котором перечисляются объекты `ZipEntry`, описывающие элементы архива.

- `ZipEntry getEntry(String name)`

  Возвращает элемент, соответствующий заданному имени, или ссылку `null`, если такого элемента нет.

  Параметр `name` Имя элемента

- `InputStream getInputStream(ZipEntry ze)`

  Возвращает объект `InputStream` для заданного элемента архива.

  Параметр `ze` Объект `ZipEntry`, описывающий элемент ZIP-файла

- `String getName()`

  Возвращает путь к данному ZIP-файлу.

Использование потоков

В следующих четырех разделах описано, как использовать преимущества некоторых представителей многочисленного семейства потоков. Будем предполагать, что мы работаем с классом `Employee` и его подклассами, например классом `Manager`. (Более подробно об этих классах см. в главах 5 и 6.) Рассмотрим четыре разных сценария, которые могут осуществляться при записи в файл массива, в котором хранится информация о сотрудниках, а затем при считывании его обратно в память.

1. Сохранение данных одного и того же типа (`Employee`) в текстовом формате.
2. Сохранение данных одного и того же типа в двоичном формате.
3. Сохранение и считывание полиморфных данных (сочетание объектов `Employee` и `Manager`).
4. Сохранение и считывание данных, содержащих встроенные ссылки (объекты `Manager` с указателями на другие объекты `Employee`).

Запись выходных данных с разделителями

Рассмотрим сохранение данных в массиве, состоящем из объектов `Employee`. Мы будем использовать широко известный *формат с разделителями* (delimited format). Каждая запись расположена в отдельной строке. Поля отделяются друг от друга разделителями. В качестве разделителя будет использоваться вертикальная черта (|). (Популярен также разделитель :.) Естественно, мы отдельно рассмотрим случай, когда разделитель содержится в одной из сохраняемых строк.

В операционной системе Unix большинство файлов хранится именно в таком формате. Нам известны случаи, когда в этом формате хранились целые базы данных о сотрудниках, содержащие тысячи записей, причем для работы с ними не требовалось ничего, кроме утилит `awk`, `sort` и `join`. (На PC, где программы для работы с базами данных достаточно дешевы, такой формат применяется намного реже.)

Ниже приведен пример подобной записи.

```
Harry Hacker|35500|1989|10|1
Carl Cracker|75000|1987|12|15
Tony Tester|38000|1990|3|15
```

Сохранить эти записи достаточно легко. Поскольку мы имеем дело с текстовым файлом, следует использовать класс `PrintWriter`. В файл записываются все поля, за которыми помещается разделитель |, а если запись является последней, за ней указывается символ \n. Затем, вспомнив, что нам нужен *класс*, реагирующий на сообщения, добавим в класс `Employee` метод `writeData()`.

```
public void writeData(PrintWriter out) throws IOException
{
   GregorianCalendar calendar = new GregorianCalendar();
   calendar.setTime(hireDay);
   out.println(name + "|"
      + salary + "|"
      + calendar.get(Calendar.YEAR) + "|"
      + (calendar.get(Calendar.MONTH) + 1) + "|"
      + calendar.get(Calendar.DAY_OF_MONTH));
}
```

При чтении записи вводятся строка за строкой, а затем из строк выделяются поля. Ниже мы рассмотрим вспомогательный класс, предусмотренный в языке Java для облегчения этой работы.

Класс StringTokenizer и текст с разделителями

При считывании данных мы получаем одну очень длинную строку. Затем ее нужно разбить на отдельные строки. Для этого необходимо найти разделитель "|" и выделить последовательности символов, содержащиеся между разделителями. (Такие последовательности называются *лексемами*.) Класс `StringTokenizer` из пакета `java.util` предназначен для выделения лексем. Он позволяет легко разбивать на части длинные строки, в которых записан текст, размеченный разделителями. Объект `StringTokenizer` присоединяется к строке. Создавая такой объект, нужно указать символ-разделитель.

```
StringTokenizer t = new StringTokenizer(line, "|");
```

В одной строке может быть несколько разделителей.

```
StringTokenizer tokenizer = new StringTokenizer(line, "|,;");
```

Каждый из символов, содержащийся в строке, которая указана в качестве второго параметра конструктора, выполняет функции разделителя.

Если вы не укажете разделитель, то по умолчанию будет принято значение " \t\n\r", т.е. в роли разделителя будет выступать либо пробел, либо знак табуляции, либо перевод строки, либо возврат каретки.

Если вы создадите объект `StringTokenizer`, то сможете использовать его методы для быстрого извлечения лексем из строки. Метод `nextToken()` возвращает очередную лексему, которая еще не была прочитана. Метод `hasMoreTokens()` возвращает значение `true`, если доступна еще хотя бы одна лексема. Приведенный ниже фрагмент кода обрабатывает все лексемы.

```
while (tokenizer.hasMoreTokens())
{
   String token = tokenizer.nextToken();
   process token
}
```

Альтернативой класса `StringTokenizer` является метод `split()` класса `String`. В результате вызова `line.split("[|,;]")` будут получены лексемы, для разделения которых использованы разделители, указанные в квадратных скобках. Для описания разделителей можно использовать любые регулярные выражения. (Регулярные выражения будут рассмотрены в конце данной главы.)

`java.util.StringTokenizer 1.0`

- `StringTokenizer(String str, String delim)`

  Создает объект `StringTokenizer`, реагирующий на указанные разделители.

Параметры	`str`	Строка, из которой считываются лексемы
	`delim`	Строка, в которой записаны символы-разделители (каждый символ в этой строке является разделителем)

- `StringTokenizer(String str)`

  Создает объект класса `StringTokenizer` с разделителями по умолчанию (" \t\n\r").

- `boolean hasMoreTokens()`

  Если в строке еще есть лексемы, возвращает значение `true`.

- `String nextToken()`

  Возвращает следующую лексему. Если лексем больше нет, генерирует исключение `NoSuchElementException`.

- `String nextToken(String delm)`

  Переключается на новое множество разделителей и после этого возвращает следующую лексему. Установленные в процессе выполнения метода разделители используются и далее.

- `int countTokens()`

  Возвращает количество лексем, оставшихся в строке.

Чтение текста с разделителями

Записи класса `Employee` считываются довольно просто: метод `readLine()` класса `BufferedReader` читает их строка за строкой. Для того чтобы поместить одну запись в отдельную строку, нужен следующий фрагмент кода:

```
BufferedReader in
   = new Bufferedreader(new FileReader("employee.dat"));
...
String line = in.readLine();
```

Затем надо выделить отдельные лексемы. После этого мы получим *строки*, которые следует преобразовать в числа.

В класс `Employee` необходимо добавить метод `readData()`, а затем вызвать его.

```
e.readData(in);
```

Этот метод уничтожает предыдущее содержимое объекта `e`, заменяя его новой строкой. Учтите, что такой метод может сгенерировать исключение `IOException`, если это сделает метод `readLine()`. Данный метод вряд ли может предпринять осмысленные действия в ответ на `IOException`, поэтому лучше всего передать исключение по цепочке вызовов для дальнейшей обработки.

Ниже приведен код метода `readData()`.

```
public void readData(BufferedReader in) throws IOException
{
   String s = in.readLine();
   StringTokenizer t = new StringTokenizer(s, "|");
   name = t.nextToken();
   salary = Double.parseDouble(t.nextToken());
   int y = Integer.parseInt(t.nextToken());
   int m = Integer.parseInt(t.nextToken());
   int d = Integer.parseInt(t.nextToken());
   GregorianCalendar calendar
      = new GregorianCalendar(y, m - 1, d);
   // В классе GregorianCalendar январь обозначается нулем.
   hireDay = calendar.getTime();
}
```

В программе, использующей этот метод, статический метод

```
void writeData(Employee[] e, PrinWriter out)
```

сначала записывает длину массива, а затем каждую запись. Статический метод

```
Employee[] readData(BufferedReader in)
```

сначала считывает длину массива, а затем читает каждую запись, как показано в листинге 12.2.

Листинг 12.2. Содержимое файла `DataFileTest.java`

```
import java.io.*;
import java.util.*;

public class DataFileTest
{
   public static void main(String[] args)
   {
      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Carl Cracker",
         75000, 1987, 12, 15);
      staff[1] = new Employee("Harry Hacker",
         50000, 1989, 10, 1);
      staff[2] = new Employee("Tony Tester",
         40000, 1990, 3, 15);

      try
      {
         // Сохранение всех записей о сотрудниках в файле
         // employee.dat
         PrintWriter out = new PrintWriter(new
            FileWriter("employee.dat"));
         writeData(staff, out);
         out.close();

         // Помещение всех записей в новый массив.
         BufferedReader in = new BufferedReader(new
            FileReader("employee.dat"));
         Employee[] newStaff = readData(in);
         in.close();

         // Вывод вновь прочитанных записей о сотрудниках.
         for (Employee e : newStaff)
            System.out.println(e);
      }
      catch(IOException exception)
      {
         exception.printStackTrace();
      }
   }

   /**
      Записывает все объекты Employee в массив
      и передает объекту PrintWriter.
      @param e Массив объектов Employee
```

```
      @param out Объект PrintWriter
   */
   static void writeData(Employee[] employees, PrintWriter out)
      throws IOException
   {
      // Запись числа сотрудников.
      out.println(employees.length);

      for (Employee e : employees)
         e.writeData(out);
   }

   /**
      Чтение массива сотрудников из объекта BufferedReader.
      @param in Объект BufferedReader
      @return Массив сотрудников
   */
   static Employee[] readData(BufferedReader in)
      throws IOException
   {
      // Определение размера массива.
      int n = Integer.parseInt(in.readLine());

      Employee[] employees = new Employee[n];
      for (int i = 0; i < n; i++)
      {
         employees[i] = new Employee();
         employees[i].readData(in);
      }
      return employees;
   }
}

class Employee
{
   public Employee() {}

   public Employee(String n, double s, int year,
      int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new
         GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }
```

```
   public Date getHireDay()
   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public String toString()
   {
      return getClass().getName()
         + "[name=" + name
         + ",salary=" + salary
         + ",hireDay=" + hireDay
         + "]";
   }

   /**
      Запись данных о сотрудниках в объект PrintWriter.
      @param out Объект PrintWriter
   */
   public void writeData(PrintWriter out) throws IOException
   {
      GregorianCalendar calendar = new GregorianCalendar();
      calendar.setTime(hireDay);
      out.println(name + "|"
         + salary + "|"
         + calendar.get(Calendar.YEAR) + "|"
         + (calendar.get(Calendar.MONTH) + 1) + "|"
         + calendar.get(Calendar.DAY_OF_MONTH));
   }

   /**
      Чтение данных о сотрудниках из объекта BufferedReader.
      @param in Объект BufferedReader
   */
   public void readData(BufferedReader in) throws IOException
   {
      String s = in.readLine();
      StringTokenizer t = new StringTokenizer(s, "|");
      name = t.nextToken();
      salary = Double.parseDouble(t.nextToken());
      int y = Integer.parseInt(t.nextToken());
      int m = Integer.parseInt(t.nextToken());
      int d = Integer.parseInt(t.nextToken());
      GregorianCalendar calendar = new
         GregorianCalendar(y, m - 1, d);
      hireDay = calendar.getTime();
   }

   private String name;
   private double salary;
   private Date hireDay;
}
```

Класс StringBuilder

При обработке входных данных часто возникает необходимость формировать строки из отдельных символов или кодовых единиц Unicode. Использовать операцию конкатенации в этих целях неэффективно. Каждый раз, когда вы добавляете к строке новый символ, виртуальной машине приходится находить новую область памяти, достаточную для хранения строки большего размера. Это занимает много времени. При каждом добавлении одного или нескольких символов строку приходится снова и снова перезаписывать. Решить эту проблему позволяет класс `StringBuilder`.

Класс `StringBuilder` аналогичен `ArrayList`. Он управляет массивом `char[]` так, что тот может увеличивать или уменьшать свои размеры. В процессе работы вы можете добавлять или удалять кодовые единицы. В конце, когда строка оказывается сформированной, можно вызвать метод `toString()` и преобразовать содержимое массива в обычный объект `String`.

Класс `StringBuilder` был реализован в JDK 5.0. Его предшественник, `StringBuffer`, работал не так эффективно, но позволял добавлять или удалять символы из нескольких потоков. Если редактирование строки должно осуществляться различными потоками, вы можете использовать вместо `StringBuilder` класс `StringBuffer`. В обоих классах содержатся одинаковые методы.

Ниже перечислены наиболее важные методы классов `StringBuilder` и `StringBuffer`.

java.lang.StringBuilder 5.0

java.lang.StringBuffer 1.0

- `StringBuilder/StringBuffer()`

  Создает пустой объект `StringBuilder` или `StringBuffer`.
- `StringBuilder/StringBuffer(int length)`

  Создает пустой объект `StringBuilder` или `StringBuffer` с заранее заданным размером массива.
- `StringBuilder/StringBuffer(String str)`

  Создает пустой объект `StringBuilder` или `StringBuffer`, инициализированный строкой `str`.
- `int length()`

  Возвращает число кодовых единиц в объекте `StringBuilder` или `StringBuffer`.
- `StringBuilder/StringBuffer append(String str)`

  Добавляет строку и возвращает ссылку `this`.
- `StringBuilder/StringBuffer append(char c)`

  Добавляет кодовую единицу и возвращает ссылку `this`.
- `StringBuilder/StringBuffer appendCodePoint(int cp)` 5.0

  Добавляет кодовую точку, преобразуя ее в одну или две кодовые единицы, и возвращает ссылку `this`.

- `void setCharAt(int i, char c)`

  Устанавливает i-ю кодовую единицу равной c.
- `StringBuilder/StringBuffer insert(int offset, String str)`

  Включает строку в позиции `offset` и возвращает ссылку `this`.
- `StringBuilder/StringBuffer insert(int offset, char c)`

  Включает кодовую единицу в позиции `offset` и возвращает ссылку `this`.
- `StringBuilder/StringBuffer delete(int startIndex, int endIndex)`

  Удаляет кодовые единицы в позициях от `startIndex` до `endIndex - 1` и возвращает ссылку `this`.
- `String toString()`

  Возвращает строку, содержащую те же данные, что и объект `StringBuilder` или `StringBuffer`.

Потоки с произвольным доступом

Если у вас много записей переменной длины, способ хранения данных, описанный в предыдущем разделе, имеет один недостаток: запись, хранящуюся в середине файла, невозможно прочитать, не читая все предыдущие записи. Будем считать, что все записи имеют одинаковую длину. Это позволит нам реализовать метод произвольного доступа с помощью уже знакомого нам класса `RandomAccessFile`. Его можно использовать для получения любой записи, причем на это всегда будет уходить одно и то же время.

В полях нашего класса будут храниться числа, записанные в двоичном формате с помощью методов `writeInt()` и `writeDouble()`, объявленных в интерфейсе `DataOutput`. (Как уже отмечалось, этот интерфейс является общим для классов `DataOutputStream` и `RandomAccessFile`.)

Однако, поскольку длина каждой записи должна оставаться постоянной, размер строк нужно сделать одинаковым. Формат UTF, символы которого имеют разный размер, для этой цели не подходит, и в библиотеке языка Java таких методов тоже нет. Поэтому нам придется написать два вспомогательных метода, которые позволят выровнять строки по заданному размеру. Назовем эти методы `writeFixedString()` и `readFixedString()`. Они будут записывать и считывать Unicode-строки так, что длина их будет одинаковой.

Метод `writeFixedString()` получает параметр `size`. Затем он записывает указанное количество символов, отсчитывая от начала строки. (Недостающие символы заменяются кодом 0.)

Исходный код метода `writeFixedString()` имеет следующий вид:

```
static void writeFizxedString(String s, int size,
   DataOutput out) throws IOException
{
   int i;
   for (i=0; i < size; i++)
   {
     char ch = 0;
     if (i < s.kength()) ch = s.charAt(i);
     out.writeChar(ch);
   }
}
```

Метод `readFixedString()` считывает из входного потока символы до тех пор, пока их количество не станет равным значению `size` или не встретится символ, имеющий код 0 в системе Unicode. Оставшиеся нулевые символы игнорируются. Для большей эффективности этот метод использует при чтении строк класс `StringBuffer`.

```
static String readFixedString(int size, DataInput in)
   throws IOException
{
   StringBuilder b = new StringBuilder(size);
   int i = 0;
   boolean more = true;
   while (more && i < size)
   {
      char ch = in.readChar();
      i++;
      if (ch == 0) more = false;
      else b.append(ch);
   }
   in.skipBytes(2 * (size - i));
   return b.toString();
}
```

Методы `writeFixedString()` и `readFixedString()` мы реализовали в составе вспомогательного класса `DataIO`.

Для того чтобы запись имела фиксированную длину, надо записать все поля в двоичном виде.

```
public void writeData(DataOutput out) throws IOException
   {
      DataIO.writeFixedString(name, NAME_SIZE, out);
      out.writeDouble(salary);

      GregorianCalendar calendar = new GregorianCalendar();
      calendar.setTime(hireDay);
      out.writeInt(calendar.get(Calendar.YEAR));
      out.writeInt(calendar.get(Calendar.MONTH) + 1);
      out.writeInt(calendar.get(Calendar.DAY_OF_MONTH));
   }
```

Прочитать данные также просто.

```
public void readData(DataInput in) throws IOException
{
   name = DataIO.readFixedString(NAME_SIZE, in);
   salary = in.readDouble();
   int y = in.readInt();
   int m = in.readInt();
   int d = in.readInt();
   GregorianCalendar calendar =
      new GregorianCalendar(y, m - 1, d);
   hireDay = calendar.getTime();
}
```

Мы указали, что имя name всегда записывается с помощью 40 символов. Запись состоит из следующих данных:

40 символов = 80 байтов для записи имени;

1 double = 8 байтов;

3 int = 12 байтов.

Таким образом, длина каждой записи равна 100 байтам.

Допустим, что нам нужно переместить указатель файла на третью запись. Используем следующий вариант метода seek():

```
long n = 3;
int RECORD_SIZE = 100;
in.seek((n - 1) & RECORD_SIZE);
```

Теперь можно прочитать запись.

```
Employee e = new Emplotee();
e.readData(in);
```

Если вы хотите изменить запись, а затем вставить ее на то же самое место, не забудьте переместить указатель на начало этой записи.

```
in.seek((n - 1) * RECORD_SIZE;
e.wtiteData(in);
```

Чтобы определить полное количество байтов в файле, используйте метод length(). Общее количество записей равно длине файла, разделенной на размер каждой записи.

```
long int nbytes = in.length(); // Длина в байтах.
int nrecords = (int)(nbytes / RECORD_SIZE);
```

Программа, исходный текст которой приведен в листинге 12.3, сохраняет в файле три записи, а затем считывает их оттуда в обратном порядке. Для того чтобы сделать это эффективно, необходим произвольный доступ; третью запись нам нужно считать в первую очередь.

Листинг 12.3. Содержимое файла RandomFileTest.java

```
import java.io.*;
import java.util.*;

public class RandomFileTest
{
   public static void main(String[] args)
   {
      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Carl Cracker",
         75000, 1987, 12, 15);
      staff[1] = new Employee("Harry Hacker",
         50000, 1989, 10, 1);
      staff[2] = new Employee("Tony Tester",
         40000, 1990, 3, 15);
```

```
      try
      {
         // Все записи о сотрудниках сохраняются
         // в файле employee.dat
         DataOutputStream out =
             new DataOutputStream(new
                FileOutputStream("employee.dat"));
         for (Employee e : staff)
            e.writeData(out);
         out.close();

         // Извлечение записей и помещение их в новый массив.
         RandomAccessFile in = new
            RandomAccessFile("employee.dat", "r");
         // Вычисление размера массива.
         int n = (int)(in.length() / Employee.RECORD_SIZE);
         Employee[] newStaff = new Employee[n];

         // Чтение записей в обратном порядке.
         for (int i = n - 1; i >= 0; i--)
         {
            newStaff[i] = new Employee();
            in.seek(i * Employee.RECORD_SIZE);
            newStaff[i].readData(in);
         }
         in.close();

         // Вывод прочитанных записей.
         for (Employee e : newStaff)
            System.out.println(e);
      }
      catch(IOException e)
      {
         e.printStackTrace();
      }
   }
}

class Employee
{
   public Employee() {}

   public Employee(String n, double s,
      int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new
         GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }
```

```
public double getSalary()
{
   return salary;
}

public Date getHireDay()
{
   return hireDay;
}

public void raiseSalary(double byPercent)
{
   double raise = salary * byPercent / 100;
   salary += raise;
}

public String toString()
{
   return getClass().getName()
      + "[name=" + name
      + ",salary=" + salary
      + ",hireDay=" + hireDay
      + "]";
}

/**
   Запись данных о сотруднике в поток вывода.
   @param out Поток вывода
*/
public void writeData(DataOutput out) throws IOException
{
   DataIO.writeFixedString(name, NAME_SIZE, out);
   out.writeDouble(salary);

   GregorianCalendar calendar = new GregorianCalendar();
   calendar.setTime(hireDay);
   out.writeInt(calendar.get(Calendar.YEAR));
   out.writeInt(calendar.get(Calendar.MONTH) + 1);
   out.writeInt(calendar.get(Calendar.DAY_OF_MONTH));
}

/**
   Чтение данных о сотруднике из потока ввода.
   @param in Поток ввода
*/
public void readData(DataInput in) throws IOException
{
   name = DataIO.readFixedString(NAME_SIZE, in);
   salary = in.readDouble();
   int y = in.readInt();
   int m = in.readInt();
   int d = in.readInt();
   GregorianCalendar calendar = new
      GregorianCalendar(y, m - 1, d);
   hireDay = calendar.getTime();
```

```
   }

   public static final int NAME_SIZE = 40;
   public static final int RECORD_SIZE =
      2 * NAME_SIZE + 8 + 4 + 4 + 4;

   private String name;
   private double salary;
   private Date hireDay;
}

class DataIO
{
   public static String readFixedString(int size, DataInput in)
      throws IOException
   {
      StringBuilder b = new StringBuilder(size);
      int i = 0;
      boolean more = true;
      while (more && i < size)
      {
         char ch = in.readChar();
         i++;
         if (ch == 0) more = false;
         else b.append(ch);
      }
      in.skipBytes(2 * (size - i));
      return b.toString();
   }

   public static void writeFixedString(String s,
      int size, DataOutput out)
      throws IOException
   {
      int i;
      for (i = 0; i < size; i++)
      {
         char ch = 0;
         if (i < s.length()) ch = s.charAt(i);
         out.writeChar(ch);
      }
   }
}
```

Объектные потоки

Формат, в котором записи имеют фиксированную длину, хорош, если данные имеют одинаковый тип. Однако объекты, создающиеся в объектно-ориентированных программах, редко относятся к одному и тому же типу. Например, массив `staff`, номинально имеющий тип `Employee`, фактически содержит объекты дочерних классов, например класса `Manager`.

При сохранении таких файлов сначала нужно записать тип каждого объекта, а затем — данные, определяющие его текущее состояние. При считывании этой информации из файла необходимо сделать следующее.

1. Прочитать тип объекта.
2. *Создать пустой объект этого типа.*
3. Заполнить его данными, хранящимися в файле.

Это можно и сделать вручную (хотя и очень утомительно). Именно так мы и поступали в первых изданиях нашей книги. Однако компания Sun Microsystems разработала мощный механизм, позволяющий проделать это намного эффективнее. Этот механизм, известный под названием *сериализация объектов* (object serialization), полностью автоматизирует описанную выше процедуру. (Позднее мы объясним, откуда происходит название "сериализация".)

Сохранение объектов различных типов

Для того чтобы сохранить данные, сначала необходимо создать объект `ObjectOutputStream`.

```
ObjectOutputStream out = new ObjectOutputStream(new
   FileOutputStream("employee.dat"));
```

Теперь сохранить объект можно с помощью метода `writeObject()` класса `ObjectOutputStream`.

```
Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1)
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15)
out.writeObject(harry);
out.writeObject(boss);
```

Для того чтобы прочитать объект из файла в память, сначала нужно получить экземпляр класса `ObjectInputStream`.

```
ObjectInputStream in = new ObjectInputStream(new
   FileInputStream("employee.dat"));
```

Затем следует извлечь объекты в том же порядке, в котором они были записаны, используя метод `readObject()`.

```
Employee e1 = (Employee)in.readObject();
Employee e2 = (Employee)in.readObject();
```

При считывании объектов из файла надо внимательно следить за их количеством, порядком и типами. Каждый вызов метода `readObject()` приводит к считыванию следующего объекта класса `Object`. Следовательно, их необходимо привести к правильному типу. Если это не нужно или вы забыли правильный тип, можно привести тип объекта к любому суперклассу или даже оставить тип `Object`. Например, переменная `e2` имеет тип `Employee`, даже если она ссылается на объект класса `Manager`. Запросить тип объекта в процессе выполнения программы можно с помощью метода `getClass()`, описанного в главе 5.

С помощью методов `writeObject()`/`readObject()` можно записывать и читать только *объекты*, а не числа. Для того чтобы записывать и читать числа, применяются методы `writeInt()`/`readInt()` или `writeDouble()`/`readDouble()`. (Классы объектных потоков реализуют интерфейсы `DataInput`/`DataOutput`.) Разумеется, числа, хранящиеся внутри объектов (например, в полях `salary` объектов `Employee`), записываются и считываются автоматически. Напомним, что в языке Java строки и масси-

вы представляют собой объекты и, следовательно, могут записываться и считываться с помощью методов `writeObject()/readObject()`.

Однако в каждый класс, предназначенный для записи и чтения из объектного потока, нужно внести одно изменение. Такой класс должен реализовывать интерфейс `Serializable`.

```
class Employee implements Serializzable (...)
```

В интерфейсе `Serializable` нет никаких методов, поэтому сам класс изменять не требуется. Это напоминает интерфейс `Cloneable`, рассмотренный в главе 6. Однако, для того, чтобы сделать класс клонируемым, приходилось переопределять метод `clone` из класса `Object`. Для сериализации класса не нужно делать *вообще ничего*.

В листинге 12.4 приведен код программы, которая записывает на диск массив, содержащий два объекта класса `Employee` и один объект класса `Manager`. Затем программа читает объекты из файла. Запись массива выполняется одним оператором.

```
Employee[] staff = new Employee[3];
...
out.writeObject(staff);
```

Чтение массива выполняется также одним оператором. Однако после считывания значение, возвращаемое методом `readObject`, необходимо привести к правильному типу.

```
Employee[] newtaff = (Employee[])in.readObject();
```

После считывания информации мы выведем данные о каждом сотруднике. В данном случае объекты, описывающие обычных сотрудников и менеджеров, легко различаются по результатам выполнения метода `toString()`. Это позволит убедиться, что мы правильно восстановили типы объектов.

Листинг 12.4. Содержимое файла `ObjectFileTest.java`

```
import java.io.*;
import java.util.*;

class ObjectFileTest
{
   public static void main(String[] args)
   {
      Manager boss = new Manager("Carl Cracker",
         80000, 1987, 12, 15);
      boss.setBonus(5000);

      Employee[] staff = new Employee[3];

      staff[0] = boss;
      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

      try
      {
         // Записи о сотрудниках сохраняются в файле employee.dat.
         ObjectOutputStream out = new ObjectOutputStream(new
            FileOutputStream("employee.dat"));
         out.writeObject(staff);
```

```
            out.close();

            // Извлечение всех записей в новый массив.
            ObjectInputStream in =  new ObjectInputStream(new
               FileInputStream("employee.dat"));
            Employee[] newStaff = (Employee[]) in.readObject();
            in.close();

            // Отображение прочитанных записей о сотрудниках.
            for (Employee e : newStaff)
               System.out.println(e);
         }
         catch (Exception e)
         {
            e.printStackTrace();
         }
      }
   }

   class Employee implements Serializable
   {
      public Employee() {}

      public Employee(String n, double s,
         int year, int month, int day)
      {
         name = n;
         salary = s;
         GregorianCalendar calendar = new
            GregorianCalendar(year, month - 1, day);
         hireDay = calendar.getTime();
      }

      public String getName()
      {
         return name;
      }

      public double getSalary()
      {
         return salary;
      }

      public Date getHireDay()
      {
         return hireDay;
      }

      public void raiseSalary(double byPercent)
      {
         double raise = salary * byPercent / 100;
         salary += raise;
      }

      public String toString()
      {
```

```
      return getClass().getName()
         + "[name=" + name
         + ",salary=" + salary
         + ",hireDay=" + hireDay
         + "]";
   }

   private String name;
   private double salary;
   private Date hireDay;
}

class Manager extends Employee
{
   /**
      @param n Имя сотрудника
      @param s Оклад
      @param year Год найма
      @param month Месяц найма
      @param day День найма
   */
   public Manager(String n, double s,
      int year, int month, int day)
   {
      super(n, s, year, month, day);
      bonus = 0;
   }

   public double getSalary()
   {
      double baseSalary = super.getSalary();
      return baseSalary + bonus;
   }

   public void setBonus(double b)
   {
      bonus = b;
   }

   public String toString()
   {
      return super.toString()
        + "[bonus=" + bonus
        + "]";
   }

   private double bonus;
}
```

API **java.io.ObjectOutputStream 1.1**

- `ObjectOutputStream(OutputStream out)`

  Создает объект `ObjectOutputStream`, после чего можно записывать объекты в указанный поток `OutputStream`.

- `void writeObject(Object obj)`

  Записывает указанный объект в поток `ObjectOutputStream`. Этот метод записывает класс объекта, сигнатуру класса, а также значение всех полей, класса и его суперклассов, определяющие состояние объекта.

`java.io.ObjectInputStream 1.1`

- `ObjectInputStream(InputStream out)`

  Создает объект `ObjectInputStream`, после чего можно считывать объекты из указанного потока `InputStream`.

- `void readObject()`

  Считывает объект из потока `ObjectInputStream`. В частности, этот метод считывает класс объекта, сигнатуру класса, а также значение всех нестатических полей класса и его суперклассов, определяющие состояние объекта. Данный метод выполняет десериализацию объекта, позволяя восстановить ссылки на объект.

Формат файла, используемый при сериализации объекта

При сериализации объекта данные записываются в особом формате. Разумеется, разработчик может использовать методы `writeObject()/readObject()` и не интересоваться точным порядком следования байтов, представляющих объект в файле. Тем не менее знать этот формат чрезвычайно полезно, поскольку он позволяет глубже понять процессы, происходящие с объектными потоками. Рассмотрим шестнадцатеричное представление содержимого файлов, в которые записаны объекты. Если вас не интересуют подробности, можете смело пропустить этот раздел.

Каждый файл начинается с 2-байтового "магического числа".

```
AC ED
```

За ним записывается номер версии формата, использованного при сериализации объекта.

```
00 05
```

(Здесь и далее двухразрядное шестнадцатеричное число соответствует одному байту.) За номером версии идет последовательность объектов в том порядке, в котором они были сохранены.

Строковые объекты записываются в следующем виде:

```
74          длина (2-байта)       символы
```

Например, строка "Harry" сохраняется в виде

```
74 00 05 Harry
```

Unicode-символы записываются в модифицированном формате UTF-8.

После самого объекта должен быть записан его класс. Дескриптор класса состоит из следующих частей.

1. Имя класса.
2. *Уникальный последовательный номер версии*, который отражает типы полей и сигнатуры методов.

3. Набор флагов, описывающих метод сериализации.
4. Дескрипторы полей данных.

Последовательный номер формируется следующим образом.

1. Упорядочиваются дескрипторы класса, суперклассов, интерфейсов, типы полей и сигнатуры методов.
2. К полученным данным применяется защищенный алгоритм хэширования (SHA – Secure Hash Algorithm).

SHA — очень быстрый алгоритм, позволяющий получить "отпечаток" крупного блока информации. Этот отпечаток всегда представляет собой 20-байтовый пакет, независимо от размера исходного блока. Он образуется в результате операций, гарантирующих, что при изменении исходных данных "отпечаток" также изменится. Алгоритм SHA стал стандартом, рекомендованным Национальным институтом науки и технологии (NIST – National Institute for Science and Technology). Однако язык Java использует в качестве "отпечатка" класса только 8 байт SHA-кода. Вероятность того, что при изменении класса его "отпечаток" также изменится, очень велика.

"Отпечатки" классов можно проверять, защищая классы от следующей последовательности событий. Объект записывается на диск; разработчик класса вносит в него изменения, например удаляет какое-то поле; старый файл читается снова. Теперь структура данных на диске не соответствует классу памяти. Попытка использовать данные в старом формате может разрушить память. Язык Java делает это событие крайне маловероятным. Для предотвращения описанной ситуации при считывании объектов надо проверять, совпадают ли "отпечатки" на диске и в памяти.

На первый взгляд может показаться, что если формат данных не изменялся, можно смело считывать объекты с диска. Однако язык Java проверяет, не изменялись ли также и методы (ведь именно методы определяют смысл данных). Разумеется, на практике может возникнуть ситуация, когда классы были изменены, а программе необходимо прочитать старые версии объектов. Эта проблема обсуждается далее в данной главе.

Идентификатор класса хранится в следующем формате:

72
2-байтовая длина имени класса
имя класса
8-байтовый отпечаток
1-байтовый флаг
2-байтовый счетчик дескрипторов полей
дескрипторы полей
78 (маркер конца)
тип суперкласса (70, если суперклассов нет)

Байт флагов состоит из 3-битовых масок, определенных в классе `java.io.ObjectStreamConstants`.

```
static final byte SC_WRITE_METHOD = 1;
   // Класс содержит метод writeObject(),
   // записывающий дополнительные данные.
```

```
static final byte SC_SERIALIZABLE = 2;
   // Класс реализует интерфейс Serializable.
static final byte SC_EXTERNALIZABLE = 3;
   // Класс реализует интерфейс Externalizable.
```

Интерфейс `Externalizable` будет рассмотрен далее в этой главе. Классы, реализующие этот интерфейс, содержат методы чтения и записи полей, определяемые пользователем. У классов, реализующих интерфейс `Serializable`, значение флага равно 02. Однако, несмотря на то, что класс `java.util.Date` использует собственные методы `readObject()/writeObject()`, его флаг равен 03.

Дескриптор поля данных имеет следующий формат:

1-байтовый код типа
2-байтовая длина имени поля
имя поля
имя класса (если поле является объектом)

Код типа может принимать одно из следующих значений:

```
B          byte
C          char
D          double
F          float
I          int
J          long
L          объект
S          short
Z          boolean
[          массив
```

Если код типа равен `L`, за именем поля записывается его тип. Класс и строки, задающие имена полей, не начинаются с кода строки 74, в отличие от типов полей. Типы полей немного иначе кодируют свои имена.

Например, поле `salary` класса `Employee` кодируется следующим образом:

```
D 00 06 salary
```

Ниже полностью приведен дескриптор класса `Employee`.

```
72 00 08 Employee
   E6 D2 86 7D AE AC 18 1B 02          "Отпечаток" и флаги
   00 03                               Количество полей
   D  00 06 salary                     Тип и имя поля
   L  00 07 hireday                    Тип и имя поля
   74 00 10 Ljava/util/Date;           Имя класса поля — String
   L  00 04 name                       Тип и имя поля
   74 00 12 Ljava/lang/String;         Имя класса поля — String
   78                                  Маркер конца
   70                                  Суперкласс отсутствует
```

Дескрипторы класса имеют довольно большую длину. Если в файле нужен еще один дескриптор *того же класса*, то используется его сокращенная форма.

71 4-байтовый последовательный номер

Указанный последовательный номер относится к предыдущему дескриптору класса. Мы обсудим данную схему нумерации несколько позже.

Объект хранится в следующем виде:

73 Дескриптор класса Данные объекта

Ниже в качестве примера приведено представление объекта `Employee`.

`40 E8 6A 00 00 00 00 00`	Значение поля `salary` — `double`
`73`	Значение поля `hireday` — новый объект
`71 00 7E 00 08`	Существующий класс `java.util.date`
`77 08 00 00 00 91 1B 4E B1 80 78`	Внешнее хранилище (см. далее)
`74 00 0C Harry Hacker`	Значение поля `name` — `String`

Как видим, в файле находится достаточно информации для восстановления объекта.

Массивы хранятся в следующем формате:

75 Дескриптор класса Количество элементов (4 байта) Элементы

В дескрипторе класса имя класса, которому принадлежит массив, представляется в том же формате, который используется платформенно-ориентированными методами. В данном случае имя класса начинается буквой `L`, а заканчивается двоеточием.

Например, массив, состоящий из трех объектов класса `Employee`, начинается следующим образом:

`75`	Массив
`72 00 0B [LEmployee;`	Новый класс, длина строки, имя класса `Employee[]`
`FC BF 36 11 C5 91 11 C7 02`	"Отпечаток" класса и флаги
`00 00`	Количество полей
`78`	Маркер конца
`70`	Суперкласс отсутствует
`00 00 00 03`	Количество элементов массива

Обратите внимание на то, что "отпечаток" массива, состоящего из объектов класса `Employee`, отличается от "отпечатка" самого класса `Employee`.

Разумеется, изучать эти коды так же интересно, как читать телефонную книгу. Однако они довольно поучительны и позволяют проследить, каким образом в объектных потоках хранятся детальные описания всех объектов, причем этой информации достаточно для восстановления как самих объектов, так и массивов, состоящих из них.

Проблемы, связанные с сохранением ссылок на объекты

Нам известно, как хранить объекты, состоящие из чисел, строк или других простых объектов. Однако мы еще не рассмотрели, что произойдет, если объект используется другими объектами и определяет их состояние.

Чтобы проиллюстрировать эту ситуацию, внесем в класс `Manager` небольшое изменение. Допустим, что каждый менеджер имеет секретаря, которому соответствует

переменная secretary типа Employee. (Для этой цели имеет смысл создать класс Secretary как подкласс класса Employee, но мы этого делать не будем.)

```
class Manager extends Employee
{
   ...
   private Employee secretary;
}
```

Сделав это, мы должны учесть, что объект класса Manager теперь содержит *ссылку* на объект класса Employee, описывающий секретаря, а не копию этого объекта.

В частности, два менеджера могут пользоваться услугами одного и того же секретаря. Это демонстрирует рис. 12.6 и следующий фрагмент кода:

```
harry = new Employee("Harry Hacker", ...);
Manager carl = new Manager("Carl Crakcer", ...);
carl.setSecretary(harry);
Manager tony = new Manager("Tony Tester", ...);
tony.setSecretary(harry);
```

Допустим теперь, что мы записываем на диск данные о сотруднике. Нам совершенно не нужно, чтобы для объекта класса Manager операции выполнялись в таком порядке.

- Сохранение данных о сотруднике.
- Сохранение данных о секретаре.

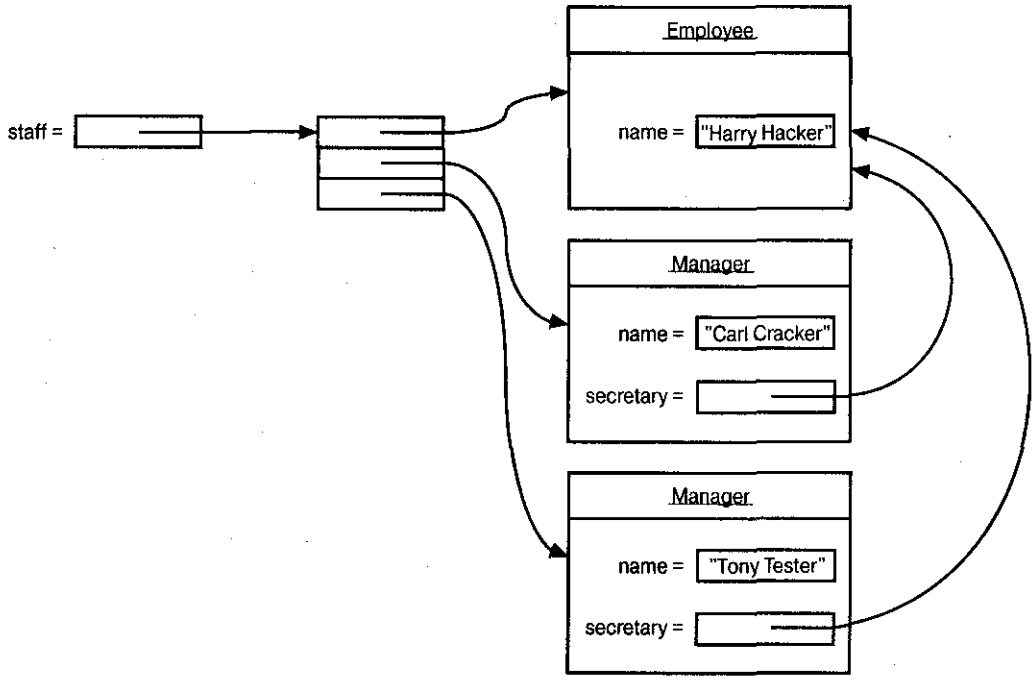


Рис. 12.6. Два менеджера с одним секретарем

Если так произойдет, то объект `harry` будет сохраняться *трижды*. При повторной загрузке объекты могут образовывать конфигурацию, показанную на рис. 12.7.

Это совсем не то, что нам нужно. Допустим, что секретарю повысили оклад. Нам совсем не хочется отслеживать все остальные копии этого объекта и вносить в них изменения. Напротив, нам хотелось бы сохранить и восстанавливать *только одну копию* объекта. Чтобы это сделать, нужно скопировать и восстановить исходные ссылки на объекты. Иными словами, мы хотим, чтобы образ объектов на диске точно совпа-

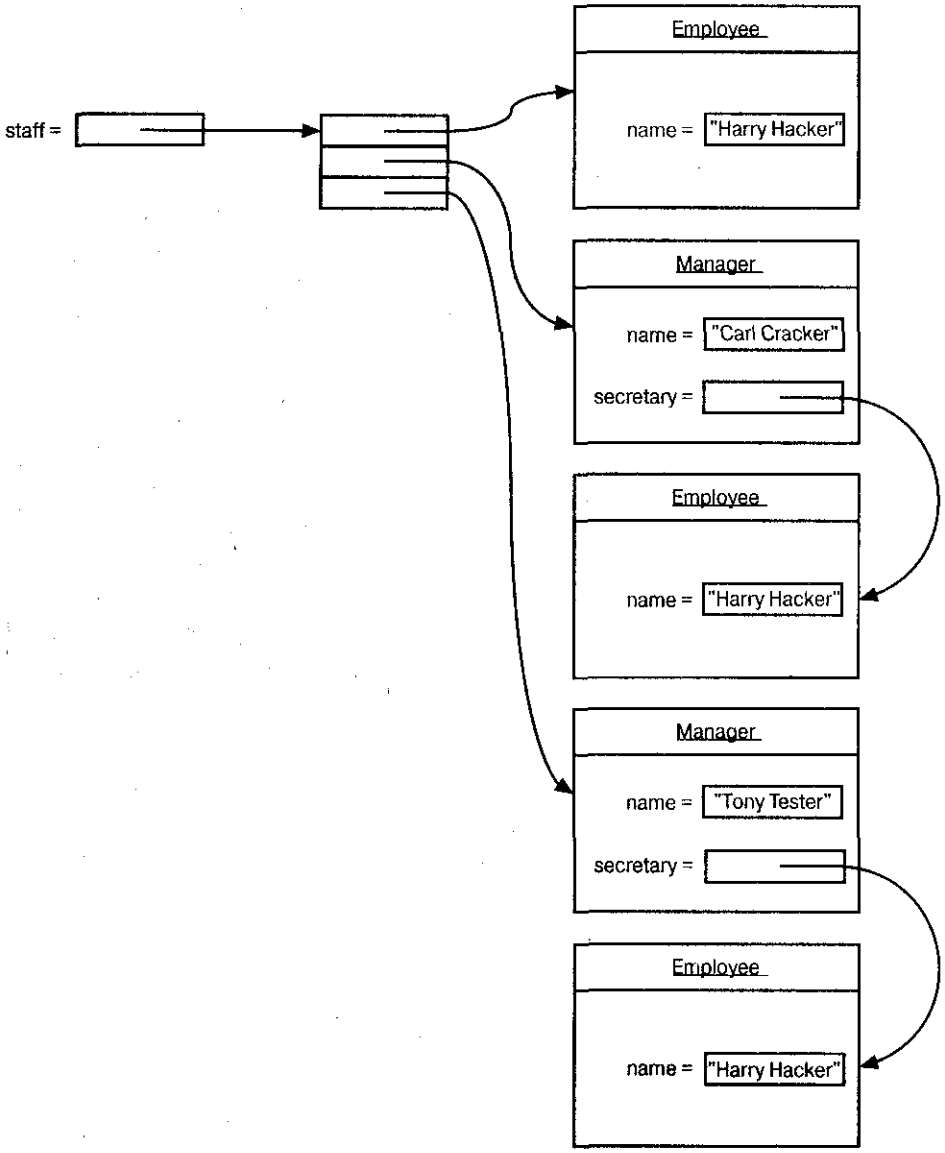


Рис. 12.7. Один из объектов сохранен три раза

дал с образом объектов в памяти. Этот эффект называется *постоянством* (persistence) в объектных циклах.

Разумеется, мы не можем сохранять и восстанавливать адреса памяти для объектов, описывающих секретарей. При повторной загрузке объект будет размещен совсем в другом месте.

Вместо этого в языке Java применяется *сериализация объектов*. Алгоритм сериализации выглядит следующим образом.

- Все объекты, сохраненные на диске, получают последовательные номера (1, 2, 3 и т.д., как показано на рис. 12.8).
- При сохранении объекта следует проверить, не сохранен ли на диске такой же объект.
- Если такой объект уже был сохранен, следует сделать заметку "аналогичен предыдущему сохраненному объекту с последовательным номером *x*". Если нет, сохраняются все данные, содержащиеся в этом объекте.

При *восстановлении объекта* процедуру следует выполнять в обратном порядке. Для каждого загружаемого объекта проверяется его номер и определяется адрес в памяти, где он записан. При обнаружении пометки "аналогичен предыдущему сохра-

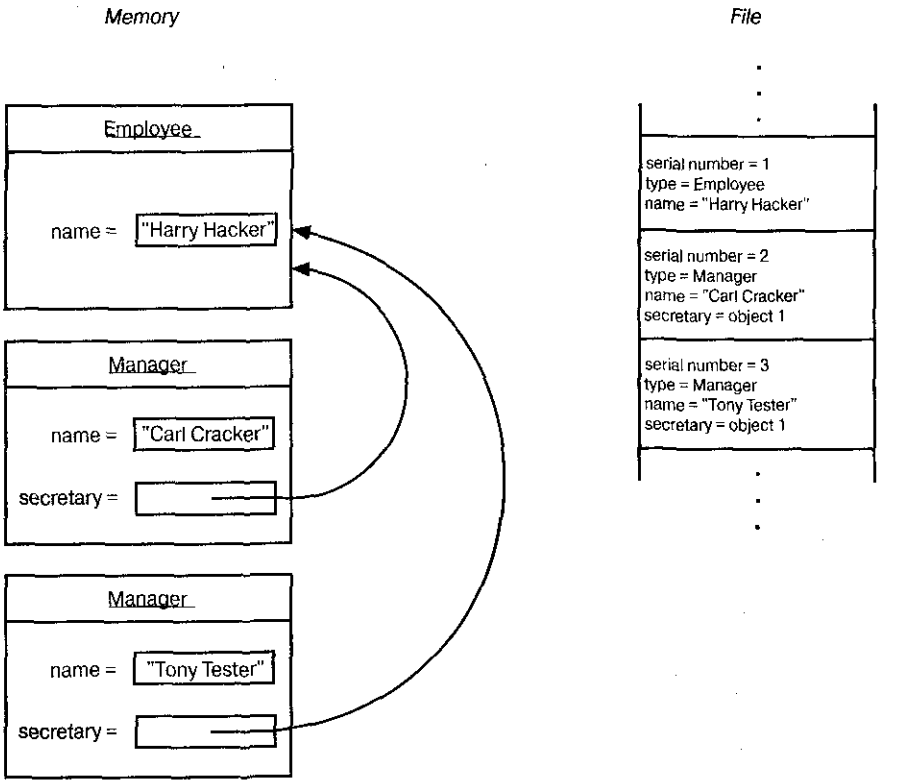


Рис. 12.8. Пример сериализации объекта

ненному объекту с последовательным номером *x*" нужно найти объект с номером *x* и создать ссылку на него.

Обратите внимание на то, что объекты необязательно сохранять в каком-то фиксированном порядке. На рис. 12.9 показано, что произойдет, если в массиве, хранящем информацию о сотрудниках, первым будет обнаружен объект `Manager`.

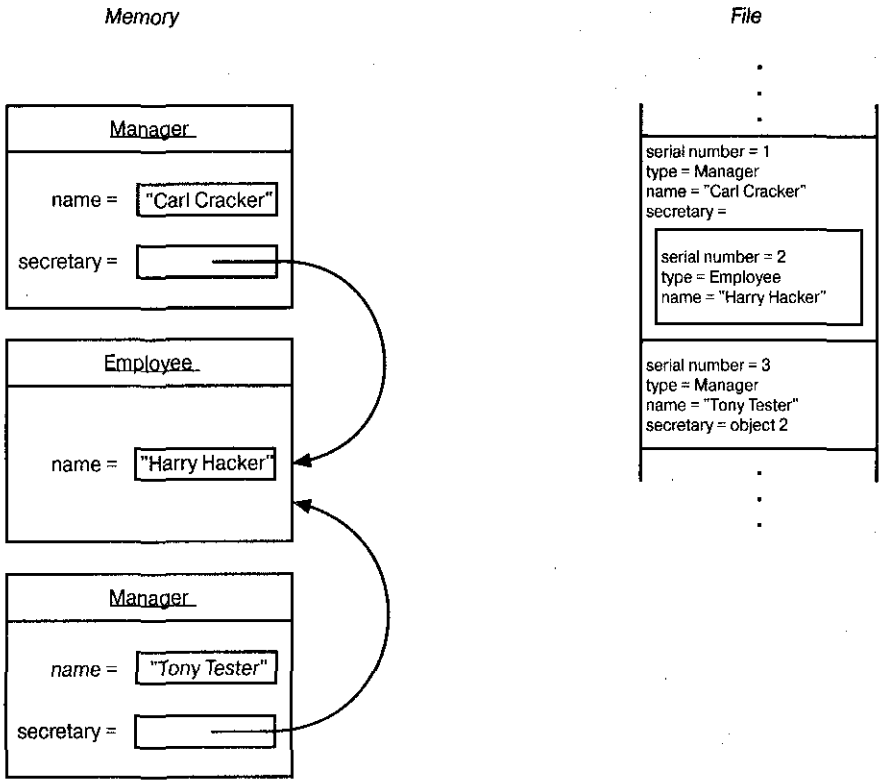


Рис. 12.9. Объекты, сохраненные в произвольном порядке

Действие рассматриваемого механизма запомнить довольно трудно. К счастью, при использовании объектных потоков процесс выполняется *полностью автоматически*. Объектные потоки присваивают последовательные номера и отслеживают дублирование объектов. Используемая на практике схема нумерации немного отличается от той, которая показана на рисунках; подробно об этом речь пойдет в следующем разделе.

В этой главе мы используем сериализацию для записи набора объектов в файл и для извлечения их оттуда в исходном виде. Другое важное применение этого механизма — передача набора объектов по сети на другой компьютер. Адреса памяти не имеют смысла, когда речь идет о файле, и они тем более бесполезны при переносе объектов на компьютер с другим процессором. При сериализации адреса заменяются последовательными номерами. Это позволяет передавать наборы объектов с одного компьютера на другой. Такое применение сериализации будет рассмотрено при обсуждении удаленных вызовов методов во втором томе.

В листинге 12.5 приведен код программы, которая передает и получает данные о сотрудниках и менеджерах по сети (некоторые из менеджеров пользуются услугами одного и того же секретаря). Обратите внимание на то, что объект, описывающий секретаря, после повторной загрузки остается единственным — когда элемент `newStaff[1]` получает повышение оклада, это отражается в полях `secretary` объектов, описывающих менеджеров.

Листинг 12.5. Содержимое файла `ObjectRefTest.java`

```
import java.io.*;
import java.util.*;

class ObjectRefTest
{
   public static void main(String[] args)
   {
      Employee harry = new Employee("Harry Hacker",
         50000, 1989, 10, 1);
      Manager boss = new Manager("Carl Cracker",
         80000, 1987, 12, 15);
      boss.setSecretary(harry);

      Employee[] staff = new Employee[3];

      staff[0] = boss;
      staff[1] = harry;
      staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

      try
      {
         // Сохранение всех записей о сотрудниках
         // в файле employee.dat
         ObjectOutputStream out = new ObjectOutputStream(new
               FileOutputStream("employee.dat"));
         out.writeObject(staff);
         out.close();

         // Извлечение всех записей в новый массив.
         ObjectInputStream in =  new ObjectInputStream(new
               FileInputStream("employee.dat"));
         Employee[] newStaff = (Employee[]) in.readObject();
         in.close();

         // Повышение оклада секретаря.
         newStaff[1].raiseSalary(10);

         // Вывод прочитанных записей о сотрудниках.
         for (Employee e : newStaff)
            System.out.println(e);
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
```

```
}

class Employee implements Serializable
{
   public Employee() {}

   public Employee(String n, double s,
      int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new
         GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public String toString()
   {
      return getClass().getName()
         + "[name=" + name
         + ",salary=" + salary
         + ",hireDay=" + hireDay
         + "]";
   }

   private String name;
   private double salary;
   private Date hireDay;
}

class Manager extends Employee
{
   /**
      Создание объекта Manager, описывающего менеджера
      без секретаря.
```

```
      @param n Имя сотрудника
      @param s Оклад
      @param year Год найма
      @param month Месяц найма
      @param day День найма
   */
   public Manager(String n, double s,
      int year, int month, int day)
   {
      super(n, s, year, month, day);
      secretary = null;
   }

   /**
      Назначение секретаря менеджеру.
      @param s Объект, описывающий секретаря
   */
   public void setSecretary(Employee s)
   {
      secretary = s;
   }

   public String toString()
   {
      return super.toString()
        + "[secretary=" + secretary
        + "]";
   }

   private Employee secretary;
}
```

Формат вывода ссылок на объекты

Продолжим обсуждение формата вывода данных в объектные потоки. Если вы не читали предыдущий раздел, эту часть главы также можно пропустить.

Все объекты (включая массивы и строки) и все дескрипторы классов получают при записи в файл последовательные номера. Этот процесс называется *сериализацией* (serialization). (Отсчет начинается с номера `00 7E 00 00`.)

Мы уже знаем, что полный дескриптор для заданного класса создается только однажды. Следующие дескрипторы только ссылаются на него. В рассмотренном выше примере вторая ссылка на класс `Day` в массиве объектов этого класса была закодирована следующим образом:

```
71 00 7E 00 08
```

Тот же самый механизм используется и для объектов. При записи ссылки на ранее сохраненный объект после числа `71` ставится последовательный номер объекта. По контексту всегда можно определить, к чему относится номер — к дескриптору класса или объекту.

Нулевая ссылка записывается следующим образом:

```
70
```

Прокомментируем выходные данные программы `ObjectRefTest` из предыдущего подраздела. Запустите программу и проанализируйте шестнадцатеричное представление файла `employee.dat`, а потом сравните с приведенными ниже данными. В данном случае для нас важны строки, содержащие ссылку на ранее сохраненный объект.

```
AC ED 00 05                                  Заголовок файла
75                                           Массив staff (серийный #1)
   72 00 0B [LEmployee;                      Новый класс, длина строки, имя класса Employee[] (последовательный номер #0)
      FC BF 36 11 C5 91 11 C7 02             "Отпечаток" и флаги
      00 00                                  Количество полей
      78                                     Маркер конца
      70                                     Суперкласс отсутствует
      00 00 00 03                            Количество элементов в массиве
   73                                        staff[0] — новый объект (последовательный номер #7)
      72 00 07 Manager                       Новый класс, длина строки, имя класса (последовательный номер #2)
         36 06 AE 13 63 8F 59 B7 02          "Отпечаток" и флаги
         00 01                               Количество полей
         L 00 09 secretary                   Тип и имя поля
         74 00 0A LEmployee                  Имя класса, определяющего тип поля, — String (последовательный номер #3)
         78                                  Маркер конца
         72 00 08 Employee                   Суперкласс — новый класс, длина строки, имя класса (последовательный номер #4)
            E6 D2 86 7D AE AC 18 1B 02       "Отпечаток" и флаги
            00 03                            Количество полей
            D 00 06 salary                   Тип и имя поля
            L 00 07 hireDay                  Тип и имя поля
            74 00 10 Ljava/util/Date;        Имя класса, определяющего тип поля, — String (последовательный номер #5)
            L 00 04 name                     Тип и имя поля
            74 00 12 Ljava/lang/String;      Имя класса, определяющего тип поля, — String (последовательный номер #6)
         78                                  Маркер конца
         70                                  Суперкласс отсутствует
```

```
   40 F3 88 00 00 00 00 00          Значение поля salary — double
   73                               Значение поля hireDay — новый
                                    объект (последовательный номер #9)
      72 00 0E java.util.Date       Новый класс, длина строки, имя класса (последовательный номер #8)
         68 6A 81 01 4B 59 74 19 03 "Отпечаток" и флаги
         00 00                     Переменных экземпляра нет
         78                        Маркер конца
         70                        Суперкласс отсутствует
      77 08                        Внешнее хранилище, количество байтов
      00 00 00 83 E9 39 E0 00      Дата
      78                           Маркер конца
   74 00 0C Carl Cracker           Значение поля name — String (последовательный номер #10)
   73                              Значение поля secretary — новый объект (последовательный номер #11)
      71 00 7E 00 04               Существующий класс (используется последовательный номер #4)
      40 E8 6A 00 00 00 00 00      Значение поля salary — double
      73                           Значение поля hireDay — новый объект (последовательный номер #12)
         71 00 7E 00 08            Существующий класс (используется последовательный номер #8)
         77 08                     Внешнее хранилище, количество байтов
         00 00 00 91 1B 4E B1 80   Дата
         78                        Маркер конца
      74 00 0C Hary Hacker         Значение поля name — String (последовательный номер #13)
71 00 7E 00 0B                     staff[1] — существующий объект (используется последовательный номер #11)
73                                 staff[2] — новый объект (последовательный номер #14)
   71 00 7E 00 04                  Существующий класс (используется последовательный номер #4)
   40 E3 88 00 00 00 00 00         Значение поля salary — double
   73                              Значение поля hireDay — новый объект (последовательный номер #15)
      71 00 7E 00 08               Существующий класс (используется последовательный номер #8)
```

```
   77 08                                  Внешнее хранилище, количество байтов
   00 00 00 94 6D 3E EC 00 00             Дата
   78                                     Маркер конца
 74 00 0B Tony Tester                     Значение поля name – String
                                          (последовательный номер #16)
```

Знать точный формат файла необязательно (разве что вы собираетесь злонамеренно модифицировать его вручную). Необходимо помнить лишь следующее.

- Объектный поток вывода содержит типы и поля всех объектов.
- Каждому объекту присваивается свой последовательный номер.
- Повторяющиеся объекты сохраняются в виде ссылок на соответствующий последовательный номер.

Модификация механизма сериализации, предусмотренного по умолчанию

Некоторые поля классов никогда не должны подвергаться сериализации. Например, целые числа, хранящиеся в дескрипторах файлов или окон, предназначены только для платформенно-зависимых методов. Такая информация заведомо бесполезна при повторной загрузке объекта или при пересылке на другой компьютер. Неверные данные, записанные в этих полях, при вызове платформенно-ориентированных методов могут привести к сбою системы. В языке Java есть простой механизм, позволяющий защитить эти поля от сериализации. Их следует пометить ключевым словом `transient`. Во время сериализации такие поля игнорируются.

Механизм сериализации можно настраивать для конкретных классов так, чтобы они осуществляли верификацию данных либо другие полезные действия. В классе, реализующем интерфейс `Serializable`, можно определить следующие методы:

```
private void readObject(ObjectInputStream in)
   throws IOException, ClassNotFoundException;
private void writeObject(ObjectOutputStream out)
  throws IOException;
```

Теперь поля не будут автоматически подвергаться сериализации, а вместо этого будут вызываться указанные выше методы.

Рассмотрим типичный пример. Многие классы, принадлежащие пакету `java.awt.geom`, не сериализуются. К таким классам относится, например, класс `Point2D.Double`. Предположим теперь, что нам нужно сериализовать класс `LabeledPoint`, в котором хранятся объекты класса `String` и `Point2D.Double`. Во-первых, необходимо пометить поле `Point2D.Double` ключевым словом `transient`, чтобы избежать возникновения исключения `NotSerializableException`.

```
public class LabeledPoint implements Serializable
{
   . . .
   private String label;
   private transient Point2D.Double point;
}
```

В методе `writeObject()` мы сначала запишем дескриптор объекта и поле `String`, вызвав метод `defaultWriteObject()`. Это специальный метод из класса

ObjectOutputStream, который можно вызывать только из метода writeObject() сериализуемого класса. Затем запишем координаты точки, используя вызов стандартных методов из класса DataOutput.

```
private void writeObject(ObjectOutputStream out)
   throws IOException
{
   out.defaultWriteObject();
   out.writeDouble(point.getX());
   out.writeDouble(point.getY());
}
```

В методе readObject() чтение выполняется в обратном порядке.

```
private void readObject(ObjectInputStream in)
   throws IOException
{
   in.defaultReadObject();
   double x = in.readDouble();
   double y = in.readDouble();
   point = new Point2D.Double(x, y);
}
```

Рассмотрим еще один пример. Класс java.util.Date предусматривает свои собственные методы readObject() и writeObject(), записывающие дату в виде количества миллисекунд, прошедших с начала эпохи (1 января 1970 года, полночь по Гринвичу). Класс Date имеет сложное внутреннее устройство, хранит объект класса Calendar и счетчик миллисекунд, что позволяет оптимизировать преобразования даты и времени. Состояние объекта класса Calendar представляет собой избыточную информацию, поэтому его не стоит сохранять.

Методы readObject() и writeObject() нужны только для чтения и записи полей данных объекта. Они не нужны для вывода данных из суперкласса или другой информации.

Класс может не полагаться на механизм сериализации, а определять свои собственные средства чтения и записи объектов. Для этого класс должен реализовывать интерфейс Externalizable. А это вынуждает нас определить два метода.

```
public void readExternal(ObjectInputStream in)
   throws IOException, ClassNotFoundException;
public void writeExternal(ObjectOutputStream out)
   throws IOException;
```

В отличие от методов readObject() и writeObject(), описанных выше, эти методы сохраняют и восстанавливают весь объект, *включая данные суперкласса*. Механизм сериализации просто записывает в поток класс объекта. В то же время при чтении объекта, реализующего интерфейс Externalizable, поток создает новый объект, используя конструктор по умолчанию, и вызывает метод readExternal(). Ниже приведена реализация этих методов для класса Employee.

```
public void readExternal(ObjectInput s)
   throws IOException
{
   name = s.readUTF();
   salary = s.readDouble();
   hireDay = new Date(s.readLong());
```

```
}

public void writeExternal(ObjectOutput s)
   throws IOException
{
  s.writeUTF(name);
  s.writeDouble(salary);
  s.writeLong(hireDay.getTime());
}
```

Сериализация выполняется довольно медленно, поскольку виртуальная машина должна проанализировать структуру каждого объекта. Если вы хотите достичь большей производительности и вам приходится часто записывать и считывать большое количество экземпляров конкретного класса, изучите подробнее интерфейс `Externalizable`. На Web-странице `http://developer.java.sun.com/developer/TechTips/txtarchive/Apr00_Stu.txt` показано, что для класса `Employee` использование средств `Externalizable` на 35–40% повышает скорость выполнения программы по сравнению со стандартной сериализацией.

В отличие от методов `readObject` и `writeObject`, которые объявлены как `private` и которые можно вызывать только с помощью механизма сериализации, методы `readExternal()` и `writeExternal()` общедоступны. В частности, метод `writeExternal()` позволяет модифицировать состояние существующих объектов.

Специальные варианты сериализации можно найти на Web-странице `http://www.absolutejava.com/serialization`.

Сериализация перечислений

Следует уделить особое внимание сериализации и десериализации объектов, считающихся уникальными. Обычно это происходит при реализации перечислений.

Если вы пользуетесь средством enum, предложенным в JDK 5.0, можете не беспокоиться о сериализации — все необходимые действия будут выполнены автоматически. *Однако, предположим, что вы занимаетесь поддержкой уже существующего кода* и в нем содержится следующий фрагмент:

```
public class Orientation
{
   public static final Orientation HORIZONTAL =
      new Orientation(1);
   public static final Orientation VERTICAL  = new Orientation(2);
   private Orientation(int v) { value = v; }
   private int value;
}
```

Подобная ситуация часто возникала тогда, когда нумерованный тип еще не был реализован в языке Java. Заметьте, что конструктор, в котором предусмотрен параметр, объявлен как `private`. Таким образом могут быть созданы только объекты `Orientation.HORIZONTAL` и `Orientation.VERTICAL`. В результате появляется возможность использовать оператор == для проверки идентичности объектов.

```
if (orientation == Orientation.HORIZONTAL) . . .
```

Существует одна особенность, о которой следует помнить в случае, если нумерованный тип реализует интерфейс Serializable. В этом случае механизм сериализации, используемый по умолчанию, не подходит. Предположим, что мы сохранили значение Orientation и прочитали его снова.

```
Orientation original = Orientation.HORIZONTAL;
ObjectOutputStream out = . . .;
out.write(value);
out.close();
ObjectInputStream in = . . .;
Orientation saved = (Orientation) in.read();
```

Теперь логическое выражение в составе приведенного ниже оператора if будет равно false.

```
if (saved == Orientation.HORIZONTAL) ...
```

Фактически значение saved является совершенно новым объектом типа Orientation и не равно ни одной из заранее определенных констант. Даже если конструктор имеет модификатор доступа private, механизм сериализации может создавать новые объекты!

Чтобы решить эту проблему, необходимо определить другой специальный метод сериализации — readResolve(). Если метод readResolve() определен, он вызывается после десериализации объекта. Этот метод должен возвращать объект, который затем становится значением, возвращаемым методом readObject(). В нашем случае метод readResolve будет проверять поле value и возвращать подходящую константу.

```
protected Object readResolve() throws ObjectStreamException
{
   if (value == 1) return Orientation.HORIZONTAL;
   if (value == 2) return Orientation.VERTICAL;
   return null; // Такая ситуация не должна возникнуть.
}
```

Если вы поддерживаете код, созданный до появления JDK 5.0, не забудьте добавить метод readResolve() во все сериализуемые перечисления.

Поддержка версий

Ранее мы показали, как следует записывать относительно небольшие наборы объектов в объектный поток. Однако все это сводилось к простой демонстрации работы программы. Допустим теперь, что вам нужно написать программу, позволяющую пользователю создавать некий документ. Этот документ состоит из абзацев, таблиц, графиков и т.п. Выполнив следующее выражение, можно вывести весь документ в виде объекта:

```
out.writeObject(doc);
```

Абзацы, таблицы и графики также можно автоматически выводить в виде объектов. Тогда один пользователь вашей программы сможет передать файл с данными другому пользователю, имеющему копию программы. Пользователь, получивший файл, может сразу загрузить весь документ, выполнив один-единственный оператор.

```
doc = (Document)in.readObject();
```

Однако нельзя забывать, что ваша программа когда-нибудь неизбежно будет модифицирована и вы станете распространять уже версию 1.1. Сможет ли версия 1.1 читать старые файлы? Смогут ли пользователи версии 1.0 читать файлы, созданные с помощью новой версии программы?

На первый взгляд кажется, что это невозможно. Если определение класса вдруг изменилось, его "отпечаток", полученный с помощью алгоритма SHA, также изменится. Как вы знаете, в этом случае объектный поток откажется считывать объекты, не соответствующие своим "отпечаткам". Однако в классе можно указать, что он является *совместимым* со своей предыдущей версией. Для этого нужно сначала получить "отпечаток" *предыдущей* версии класса. Это можно сделать с помощью утилиты `serialver` из пакета JDK. Например, при вызове

```
serialver Employee
```

выводится следующая информация:

```
Employee: static final long serialVerionID = -1814239825517340645L;
```

Если при запуске программы `serialver` задать опцию `-show`, то программа выведет на экран диалоговое окно (рис. 12.10).

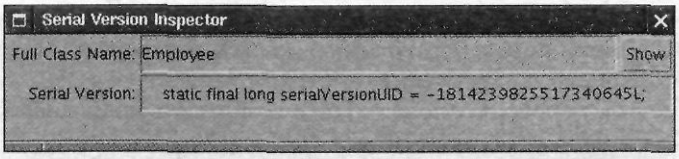


Рис. 12.10. Диалоговое окно программы `serialver`

В более поздних версиях класса нужно определять константу `serialVersionUID`, значение которой должно совпадать с "отпечатком" первоначальной версии класса.

```
class Employee implements Serializable // Версия 1.1
{
   . . .
   public static final long serialVersionUID = -1814239825517340645L;
}
```

Если в классе есть статическая переменная `serialVersionUID`, он не будет вычислять новый "отпечаток", а станет использовать старый. Теперь система сериализации сможет считывать разные версии объектов этого класса.

Если в классе изменились только методы, то проблем, связанных с чтением объектов новой версии класса, вероятнее всего, не будет. Однако, если изменению подверглись поля, эти проблемы могут возникнуть. Например, старый объект, содержащийся в файле, может содержать больше или меньше полей данных, чем объект, используемый в программе. Кроме того, в разных версиях могут отличаться типы полей. В этом случае объектный поток попытается привести находящийся в нем объект в соответствие с текущей версией класса.

Объектный поток сравнивает поля данных текущей версии класса с полями данных версии, находящейся в потоке. Разумеется, объектный поток анализирует только нестатические поля и поля, не объявленные как `transient`. Если два поля имеют одинаковые имена, но отличаются по типу, объектный поток не делает попыток пре-

образовать один тип в другой — объекты немовместимы. Если объект, находящийся в потоке, содержит поля, которых нет в текущей версии класса, эти поля игнорируются. Если текущая версия класса содержит поля, которых нет в объекте, находящемся в потоке, значения этих полей задаются по умолчанию (`null`— для объектов, 0 — для чисел и `false` — для логических значений).

Рассмотрим пример. Предположим, нам нужно записать на диск большое количество записей о сотрудниках, используя исходную версию (1.0) класса `Employee`. В данный момент класс `Employee` модифицирован, и существует его версия 2.0, в которой добавлено поле `department`. На рис. 12.11 показано, что произойдет, если объект версии 1.0 будет читаться программой, использующей объекты версии 2.0. В поле `department` будет записана ссылка `null`. На рис. 12.12 показана противоположная ситуация: программа, использующая объекты версии 1.0, читает объекты версии 2.0. Дополнительное поле `department` при этом игнорируется.

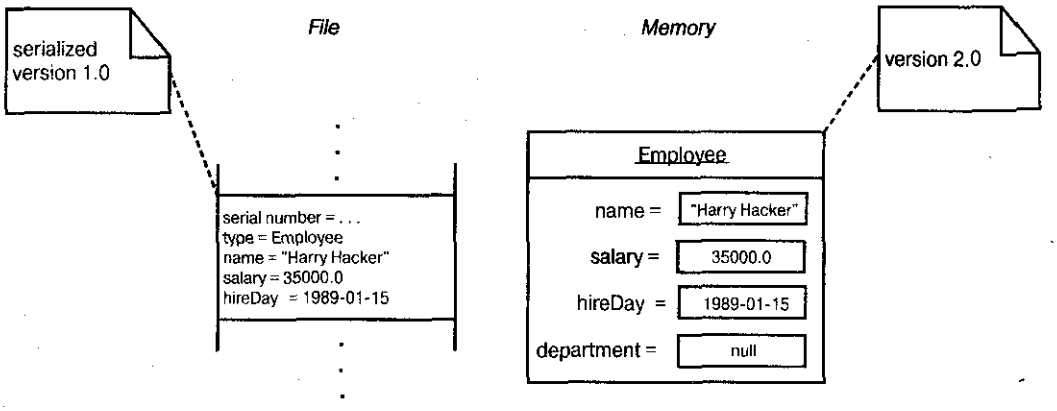


Рис. 12.11. Чтение объекта, содержащего меньшее количество полей

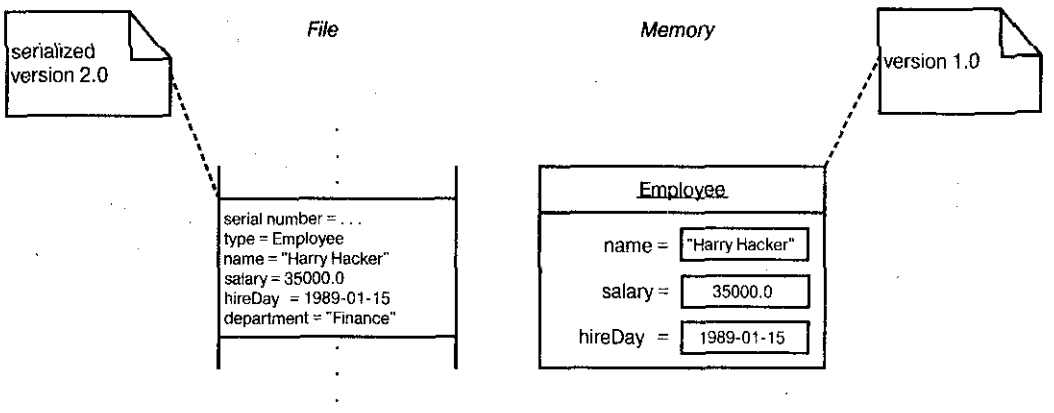


Рис. 12.12. Чтение объекта, содержащего большее количество полей

Безопасен ли этот процесс? Не всегда. Игнорирование дополнительных полей данных выглядит безобидно — программа, получившая такой объект, продолжает успешно работать со всеми полями, которые ей уже известны. Однако записывать ссылку `null` в дополнительное поле довольно опасно. Многие конструкторы специально разрабатываются для того, чтобы инициализировать поля классов ненулевыми значениями, поэтому методы таких классов могут оказаться не готовыми к работе с нулевыми ссылками. Разработчик класса вынужден либо вносить в метод `readObject()` дополнительный код, решающий проблему несовместимости версий, либо создавать методы, устойчиво работающие с нулевыми ссылками.

Применение сериализации для клонирования

Механизм сериализации бывает очень полезен при клонировании экземпляров классов, реализующих интерфейс `Serializable`. (Напомним, что в главе 6 мы уже рассматривали клонирование и отметили, что для его осуществления нужно выполнить довольно много работы.)

Для того чтобы клонировать сериализуемый объект, его нужно записать в поток вывода, а затем считать обратно. В результате возникнет новый объект, представляющий собой полную копию существующего объекта. Объект не обязательно записывать в файл — можно применить `ByteArrayOutputStream` и сохранить данные в виде последовательности байтов.

В листинге 12.6 показано, как легко создать метод `clone()` — для этого достаточно определить подкласс класса `SerialCloneable`.

Об этом способе клонирования следует знать, хотя очевидно, что он работает медленнее, чем механизм непосредственного клонирования, в котором новый объект конструируется явным образом, а данные, хранящиеся в полях оригинала, копируются (см. главу 6).

Листинг 12.6. Содержимое файла `SerialCloneTest.java`

```
import java.io.*;
import java.util.*;

public class SerialCloneTest
{
   public static void main(String[] args)
   {
      Employee harry = new Employee("Harry Hacker", 35000,
         1989, 10, 1);
      // Клонирование объекта harry.
      Employee harry2 = (Employee) harry.clone();

      // Изменение объекта harry.
      harry.raiseSalary(10);

      // Теперь объект harry отличается от его клона.
      System.out.println(harry);
      System.out.println(harry2);
   }
}

/**
```

```
   Класс, в котором реализован метод clone(),
   использующий сериализацию.
*/
class SerialCloneable implements Cloneable, Serializable
{
   public Object clone()
   {
      try
      {
         // Сохранение объекта в байтовом массиве.
         ByteArrayOutputStream bout = new ByteArrayOutputStream();
         ObjectOutputStream out = new ObjectOutputStream(bout);
         out.writeObject(this);
         out.close();

         // Чтение клона объекта из байтового массива.
         ByteArrayInputStream bin = new
            ByteArrayInputStream(bout.toByteArray());
         ObjectInputStream in = new ObjectInputStream(bin);
         Object ret = in.readObject();
         in.close();

         return ret;
      }
      catch (Exception e)
      {
         return null;
      }
   }
}

/**
   Класс Employee, определенный как подкласс
   класса SerialCloneable.
*/
class Employee extends SerialCloneable
{
   public Employee(String n, double s,
      int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new
         GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }
```

```
   public Date getHireDay()
   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public String toString()
   {
      return getClass().getName()
         + "[name=" + name
         + ",salary=" + salary
         + ",hireDay=" + hireDay
         + "]";
   }

   private String name;
   private double salary;
   private Date hireDay;
}
```

Работа с файлами

Вы уже знаете, как читать данные из файла и записывать их в файл. Однако работа с файлами не сводится лишь к чтению и записи. Более широкие функциональные возможности, позволяющие работать с файловой системой на компьютере пользователя, реализованы в классе `File`. Например, с помощью этого класса можно определять время последней модификации файлов, а также удалять или переименовывать их. Иными словами, потоковые классы сосредоточивают внимание на содержимом файлов, а класс `File` — на самом файле, хранящемся на диске.

Как это часто бывает в языке Java, возможности класса `File` реализовывались по принципу "наименьшего общего знаменателя". Например, в системе Windows можно распознать (или установить) признак "только для чтения". Но хотя вы и можете определить, является ли файл скрытым, чтобы скрыть его, приходится прибегать к помощи платформенно-зависимых методов (подробно этот вопрос будет рассматриваться во втором томе).

Простейший конструктор `File` получает полное имя файла. Если полный путь к файлу не указан, поиск проводится в текущем каталоге. (Текущим считается каталог, из которого был запущен процесс виртуальной машины. Если запуск виртуальной машины производился из командной строки, то это будет тот каталог, который был текущим на момент вызова программы java.) Например, в результате выполнения следующего выражения возвращается объект `File`, связанный с файлом, имеющим указанное имя.

```
File f = new File("text.txt");
```

Если файла с таким именем не существует, объект не создается. Фактически создание файла из объекта класса `File` осуществляется с помощью конструктора одного из потоков или метода `createNewFile()` класса `File`. Этот метод создает новый файл только в том случае, если файл с таким именем не существует, возвращая значение `true`, если работа завершена успешно, и `false` — в противном случае.

С другой стороны, имея объект `File`, можно вызвать метод `exists()`, который определяет, существует ли файл с таким именем. Например, при выполнении приведенного ниже фрагмента кода практически на любом компьютере метод `exists()` будет возвращать значение `false` и даже выводить путь к несуществующему файлу.

```
import java.io.*;

public class Test
{
   public static void main(String args[])
   {
      File f = new File("afilethatprobablydoesntexist");
      System.out.printl(f.getAbsolutePath());
      System.out.println(f.exists());
   }
}
```

В классе `File` есть еще два конструктора. Конструктор `File(String path, String name)` создает объект `File`, который соответствует файлу с заданным именем, находящемуся в каталоге, указанном посредством параметра `path`. (Если значение `path` равно `null`, конструктор создает объект `File` для файла в текущем каталоге.)

Конструктор `File(File dir, String name)` позволяет использовать уже существующий объект `File`, представляющий каталог, имя которого задано параметром `dir`. Если значение этого параметра равно `null`, конструктор создает объект `File` для файла в текущем каталоге.

Кому-то может показаться странным, но объект `File` может представлять либо файл, либо каталог (возможно, потому, что в операционной системе, которую лучше всего знают разработчики языка Java, каталоги являются файлами). Определить, что именно представляет объект `File` — файл или каталог, можно с помощью методов `isDirectory()` и `isFile()`. Это несколько неожиданно, ведь в любой объектно-ориентированной системе логично было бы иметь отдельный класс `Directory`, например расширяющий класс `File`.

Для того чтобы создать объект, представляющий каталог, достаточно задать имя каталога в конструкторе класса `File`.

```
File tempDir = new Fle(File.separator + "temp");
```

Если каталога с таким именем не существует, его можно создать с помощью метода `mkdir()`.

```
tempDir.mkdir();
```

Если объект класса `File` представляет собой каталог, массив имен файлов, находящихся в нем, можно получить, вызвав метод `list()`. Программа, приведенная в листинге 12.7, использует рассмотренные выше методы для вывода структуры каталога, имя которого задается в командной строке. (На основе этой программы легко создать класс, позволяющий получить вектор подкаталогов для дальнейшей обработки.)

Обрабатывая файлы или каталоги, всегда используйте вместо строковых имен объекты класса `File`. Например, метод `equals()` класса `File` знает, что некоторые системы не различают прописные и строчные буквы и что завершать имя каталога символом / бессмысленно.

Листинг 12.7. Содержимое файла `FindDirectories.java`

```
import java.io.*;

public class FindDirectories
{
   public static void main(String[] args)
   {
      // Если параметры не заданы, обработка
      // начинается с родительского каталога.
      if (args.length == 0) args = new String[] { ".." };

      try
      {
         File pathName = new File(args[0]);
         String[] fileNames = pathName.list();

         // Получение имен файлов из каталога.
         for (int i = 0; i < fileNames.length; i++)
         {
            File f = new File(pathName.getPath(), fileNames[i]);

            // Если файл является каталогом, рекурсивно
            // вызывается метод main.
            if (f.isDirectory())
            {
               System.out.println(f.getCanonicalPath());
               main(new String [] { f.getPath() });
            }
         }
      }
      catch(IOException e)
      {
         e.printStackTrace();
      }
   }
}
```

Вместо того, чтобы перечислять все файлы в каталоге, можно задать в качестве параметра метода `list()` объект `FileNameFilter`. Это позволит сократить список. Объекты `FileNameFilter` представляют собой экземпляры любого класса, реализующего интерфейс `FilenameFilter`.

Во всех классах, реализующих интерфейс `FileNameFilter`, должен быть определен метод `accept()`. Рассмотрим простой пример класса `FilenameFilter`, распознающего только файлы с указанным расширением.

```
public class ExtensionFlter implements FilenameFilter
{
   public ExtensionFilter(String ext)
```

```
    {
       extension = "." + ext;
    }

    public boolean accept(File dir, String name)
    {
       return name.endsWith(extension);
    }

    private String extension;
}
```

Для платформенно-независимых программ подкаталоги представляют определенную сложность. Как уже указывалось, в разных операционных системах используются разные разделители. Например, в системе Unix применяется косая черта, она же распознается и в Windows. В других системах могут быть иные соглашения, поэтому мы не рекомендуем применять косую черту.

Если, работая на платформе Windows и создавая объекты класса `File`, вы используете косую черту, метод `getAbsolutePath()` будет возвращать имя файла, в котором также будет присутствовать данный символ. Для пользователей системы Windows это может выглядеть странно. Вместо этого метода следует вызывать `getCanonicalPath()`, заменяющий косые черты обратными косыми.

Лучше всего использовать текущий разделитель каталогов, который хранится в классе `File` в статическом поле `separator`. (В среде Windows — это обратная косая черта (\), а в системе Unix — обычная косая черта (/).)

```
File foo = new File("Documents" + File.separator + "data.txt");
```

Разумеется, при использовании второго варианта конструктора класса `File` система сама определит правильный разделитель файлов.

```
File foo = new File("Documents", "data.txt");
```

Ниже приведены сведения о наиболее важных методах класса `File`.

java.io.File 1.0

- `boolean canRead()`

  Определяет, может ли текущее приложение прочитать файл.

- `boolean canWrite()`

  Определяет, можно ли записать данные в файл, или он предназначен только для чтения.

- `static boolean createTempFile(String prefix, String suffix)` **1.2**
- `static boolean createTempFile(String prefix, String suffix, File directory)` **1.2**

Создает временный файл во временном каталоге, используемом системой по умолчанию, или в каталоге, указанном пользователем. Суффикс и префикс используются для создания имени временного файла.

Параметры		
	`prefix`	Префикс, состоящий по крайней мере из трех символов
	`suffix`	Необязательный суффикс. Если задано значение `null`, используется расширение `tmp`
	`directory`	Каталог, в котором создается файл. Если задано значение `null`, используется текущий каталог

- `boolean delete()`

  Пытается удалить файл. Если файл был успешно удален, возвращает значение `true`, в противном случае — `false`.

- `void deleteOnExit()`

  Указывает, что при завершении работы виртуальной машины файл должен быть удален.

- `boolean exists()`

  Возвращает значение `true`, если файл или каталог существует, в противном случае возвращает значение `false`.

- `String getAbsolutePath()`

  Возвращает строку, содержащую абсолютный путь к файлу. Рекомендуется вместо этого метода применять метод `getCanonicalPath()`.

- *`File getCanonicalFile()`* **1.2**

  Возвращает объект `File`, содержащий каноническое имя пути к файлу. В частности, удаляются необязательные каталоги `"."`, используется правильный разделитель каталогов и регистр символов, поддерживаемый операционной системой.

- `String getCanonicalPath()` **1.1**

  Возвращает строку, содержащую каноническое имя пути к файлу. В частности, удаляются необязательные каталоги `"."`, используется правильный разделитель каталогов и регистр символов, поддерживаемый операционной системой.

- `String getName()`

  Возвращает строку, содержащую имя файла, соответствующее объекту `File` (информация о пути не включается).

- `String getParent()`

  Возвращает строку, содержащую имя родительского объекта класса `File`. Если объект представляет файл, то родительским объектом считается каталог, содержащий данный файл. Если объект представляет каталог, то родительским классом является родительский каталог. Если родительского каталога нет, возвращается значение `null`.

- `File getParentFile()` **1.2**

Возвращает объект класса `File` для родительского каталога. Понятие "родительский" определяется так же, как и для метода `getParent()`.

- `String getPath()`

  Возвращает строку, содержащую путь к файлу.

- `boolean isDirectory()`

  Возвращает значение `true`, если объект `File` представляет каталог, в противном случае возвращает значение `false`.

- `boolean isFile()`

  Возвращает значение `true`, если объект `File` представляет файл, в противном случае возвращает значение `false`.

- `boolean isHidden()` **1.2**

  Возвращает значение `true`, если объект `File` представляет скрытый файл или каталог, в противном случае возвращает значение `false`.

- `long lastModified()`

  Возвращает время последней модификации файла. Время измеряется в миллисекундах, прошедших от начала эпохи (полночь 1 января 1970 по Гринвичу). Метод возвращает число 0, если файла не существует. Для преобразования числа миллисекунд в дату используется конструктор `Date(long)`.

- `long length()`

  Возвращает длину файла в байтах, или число 0, если файла не существует.

- `String[] list()`

  Возвращает массив строк, содержащих имена файлов и каталогов, содержащихся в каталоге, который представляет объект `File`. Если объект представляет не каталог, возвращает ссылку `null`.

- `String[] list(FilenameFilter filter)`

  Возвращает массив строк, содержащих имена файлов и каталогов, содержащихся в каталоге, который представляет объект `File`, и удовлетворяющих требованиям фильтра. Если объект представляет не каталог, возвращает ссылку `null`.

  Параметры `filter` Объект `FilenameFilter`

- `File[] listFiles()` **1.2**

  Возвращает массив объектов `File`, соответствующих файлам и каталогам, содержащимся в каталоге, который представляет объект `File`. Если объект представляет не каталог, возвращает ссылку `null`.

- `File[] listFiles(FilenameFilter filter)` **1.2**

  Возвращает массив объектов `File`, соответствующих файлам и каталогам, содержащимся в каталоге, который представляет объект `File`, и удовлетворяющим требованиям фильтра. Если объект представляет не каталог возвращает ссылку `null`.

  Параметр `filter` Объект `FilenameFilter`

- `static File[] listRoots()` **1.2**

  Возвращает массив объектов `File`, представляющих доступные логические устройства. (Например, в системе Windows этот метод возвращает объекты, соответствующие локальным и сетевым устройствам. В системе Unix возвращается корневой каталог /.)

- `boolean createNewFile()` **1.2**

  Если файл с именем, соответствующим объекту `File`, отсутствует, автоматически создает новый файл. Проверка имени файла и его создание системой не прерывается другими процессами. Если файл успешно создан, возвращается значение `true`.

- `boolean mkdir()`

  Создает каталог, имя которого задается объектом класса `File`. Если каталог успешно создан, возвращает значение `true`. Если создать каталог невозможно, возвращает значение `false`.

- `boolean mkdirs()`

  В отличие от метода `mkdir` создает при необходимости родительские каталоги. Если создать такие каталоги невозможно, возвращает значение `false`.

- `boolean renameTo(File dest)`

  Переименует файл. Возвращает значение `true`, если имя было изменено, *в противном случае возвращает значение* `false`.

  Параметр `dest` Объект `File`, задающий новое имя

- `boolean setLastModified(long time)` **1.2**

  Задает время последней модификации файла. Если операция завершена успешно, возвращает значение `true`, в противном случае возвращает значение `false`.

  Параметр `time` Значение `long`, содержащее количество миллисекунд, прошедших с начала эпохи (полночь 1 января 1970 по Гринвичу). Для вычисления этого числа используется метод `getTime()` класса `Date`

- `boolean setReadOnly()` **1.2**

  Устанавливает признак "только для чтения". Если операция завершена успешно, возвращает значение `true`, в противном случае возвращает значение `false`.

- `URL toURL()` **1.2**

  Преобразует объект `File` *в объект* URL.

`java.io.FilenameFilter` 1.0

- `boolean accept(Fle dir, String name)`

  Возвращает значение `true`, если файл соответствует требованиям фильтра.

  Параметры `dir` Объект `File`, представляющий каталог, в котором содержится файл

  `name` Имя файла

Новые средства ввода-вывода

Пакет JDK 1.4 содержит большое количество усовершенствованных средств обмена данными. Набор этих средств называется "новые средства ввода-вывода" и хранится в пакете `java.nio`. (Разумеется, слово "новые" выбрано не совсем удачно – через несколько лет пакет больше не будет новым.)

Пакет обеспечивает следующие возможности.

- Поддержка виртуальной памяти.
- Блокировка файлов.
- Шифрация и дешифрация наборов символов.
- Поддержка неблокирующего ввода-вывода.

Мы обсудим только первые три свойства. Неблокирующие средства ввода-вывода, связанные с использованием потоков, будут рассмотрены во втором томе.

Поддержка виртуальной памяти

Большинство операционных систем позволяют отображать файл или его часть в память. В этом случае к содержимому файла можно обращаться как к массиву, что обеспечивает более высокую скорость по сравнению с традиционными способами обмена файлами.

В конце этого раздела мы рассмотрим программу, вычисляющую контрольную сумму CRC32-файла. При этом обращение к файлу осуществляется традиционными способами и посредством виртуальной памяти. Результаты, полученные различными способами на одной и той же машине при обработке файла `rt.jar` объемом 37 Мбайт, показаны в табл. 12.7.

Таблица 12.7. Время обработки данных при разных способах обращения к файлу

Способ обращения к файлу	Время
Обычный поток ввода	110 с
Буферизованный поток ввода	9,9 с
Произвольный доступ к файлу	162 с
Отображение файла в память	7,2 с

Разумеется, эти результаты на других машинах окажутся иными, но в любом случае отображение файла в память дает более высокие результаты по сравнению с буферизованным потоком ввода, а тем более с методом произвольного доступа к файлу.

Пользуясь пакетом `java.nio`, выполнить отображение файла в память несложно. Для этого достаточно сделать следующее.

Во-первых, нужно получить *канал* (channel) для файла. Канал – это абстракция, имеющая отношение к файлам, хранящимся на диске. Она упрощает обращение к таким функциям операционной системы, как организация отображения памяти, блокирование файлов и быстрый обмен данными между файлами. Получить канал можно, вызвав метод `getChannel()`, который был добавлен к классам `FileInputStream`, `FileOutputStream` и `RandomAccessFile`.

```
FileInputStream in = new FileInputStream( ... );
FileChannel channel = in.getChannel();
```

Во-вторых, вызвав метод `map()` класса `FileChannel`, нужно получить объект `MappedByteBuffer`, соответствующий созданному каналу. Далее следует указать, какую область файла необходимо отобразить в память, а также *режим отображения* (*mapping mode*). *Язык Java поддерживает три режима отображения файлов.*

- `FileChannel.MapMode.READ_ONLY`. В результате создается буфер, предназначенный только для чтения. Любая попытка записать в него какие-либо данные вызовет исключение `ReadOnlyBufferException`.
- `FileChannel.MapMode.READ_WRITE`. В результате создается буфер, предназначенный для записи, и в определенный момент времени все изменения записываются обратно в файл. Обратите внимание на то, что другие программы, отобразившие тот же самый файл, могут не сразу обнаружить эти изменения. Конкретные особенности одновременного отображения файла несколькими программами зависит от операционной системы.
- `FileChannel.MapMode.PRIVATE`. В результате создается буфер, предназначенный для записи, но любые изменения вносятся только в буфере и *не возвращаются* в файл.

Создав буфер, можно считывать и записывать данные, используя методы класса `ByteBuffer` и суперкласса `Buffer`.

Буферы обеспечивают как последовательный, так и произвольный доступ к данным. В буфере определена *позиция*, которая может изменяться в результате выполнения методов `get()` и `put()`. Например, с помощью следующего фрагмента кода можно просмотреть все байты в буфере:

```
while (buffer.hasRemaining())
{
   byte b = buffer.get();
   ...
}
```

Кроме того, можно использовать произвольный доступ.

```
for (int i = 0; i < buffer.limit(); i++)
{
   byte b = buffer.get(i);
   ...
}
```

Можно также прочитать массив байтов с помощью методов `get()`.

```
get(byte[] bytes)
get(byte[], int offset, int length)
```

Отметим также методы `getInt()`, `getLong()`, `getShort()`, `getChar()`, `getFloat()` и `getDouble()`, которые позволяют читать значения простых типов, хранящиеся в файлах в *двоичном* виде. Однако, если вам надо обрабатывать данные из файла, хранящиеся в порядке little-endian, необходимо вызывать следующий метод:

```
buffer.order(ByteOrder.LITTLE_ENDIAN);
```

Чтобы определить порядок, принятый в буфере в данный момент, необходимо вызвать метод `buffer.order()`.

```
ByteOrder b = buffer.order()
```

Рассмотренные методы не используют соглашение об именовании `set()/get()`.

Для записи чисел в буфер используется один из следующих методов: `pitInt()`, `putLong()`, `putShort()`, `putChar()`, `putFloat()` и `putDouble()`. В листинге 12.8 приведен код программы, вычисляющей 32-битовую контрольную сумму (CRC32). Контрольная сумма часто используется для распознавания повреждений файлов. Пакет `java.util.zip` содержит класс `CRC32`, вычисляющий контрольную сумму последовательности байтов с помощью следующего цикла:

```
CRC32 crc = new CRC32();
while (байты_не_исчерпаны)
   crc.update(следующий_байт)
long checksum = crc.getValue();
```

Подробное описание алгоритма CRC представлено на Web-странице `http://www.relisoft.com/Science/CrcMath.html`.

Детали вычислений контрольной суммы не важны. Мы просто использовали ее в качестве примера для демонстрации операций с файлами.

Запустите программу следующим образом:

```
java NIOTest имя_файла
```

Листинг 12.8. Содержимое файла `NIOTest.java`

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.zip.*;

/**
   Данная программа вычисляет контрольную сумму CRC для файла.
   Использование: java NIOTest имя_файла
*/
public class NIOTest
{
   public static long checksumInputStream(String filename)
      throws IOException
   {
      InputStream in = new FileInputStream(filename);
      CRC32 crc = new CRC32();

      int c;
      while((c = in.read()) != -1)
         crc.update(c);
      return crc.getValue();
   }

   public static long checksumBufferedInputStream(String filename)
      throws IOException
   {
      InputStream in = new BufferedInputStream(new
```

```
      FileInputStream(filename));
   CRC32 crc = new CRC32();

   int c;
   while((c = in.read()) != -1)
      crc.update(c);
   return crc.getValue();
}

public static long checksumRandomAccessFile(String filename)
   throws IOException
{
   RandomAccessFile file = new RandomAccessFile(filename, "r");
   long length = file.length();
   CRC32 crc = new CRC32();

   for (long p = 0; p < length; p++)
   {
      file.seek(p);
      int c = file.readByte();
      crc.update(c);
   }
   return crc.getValue();
}

public static long checksumMappedFile(String filename)
   throws IOException
{
   FileInputStream in = new FileInputStream(filename);
   FileChannel channel = in.getChannel();

   CRC32 crc = new CRC32();
   int length = (int) channel.size();
   MappedByteBuffer buffer = channel.map
     (FileChannel.MapMode.READ_ONLY, 0, length);

   for (int p = 0; p < length; p++)
   {
      int c = buffer.get(p);
      crc.update(c);
   }
   return crc.getValue();
}

public static void main(String[] args)
   throws IOException
{
   System.out.println("Input Stream:");
   long start = System.currentTimeMillis();
   long crcValue = checksumInputStream(args[0]);
   long end = System.currentTimeMillis();
   System.out.println(Long.toHexString(crcValue));
   System.out.println((end - start) + " milliseconds");

   System.out.println("Buffered Input Stream:");
   start = System.currentTimeMillis();
```

```
        crcValue = checksumBufferedInputStream(args[0]);
        end = System.currentTimeMillis();
        System.out.println(Long.toHexString(crcValue));
        System.out.println((end - start) + " milliseconds");

        System.out.println("Random Access File:");
        start = System.currentTimeMillis();
        crcValue = checksumRandomAccessFile(args[0]);
        end = System.currentTimeMillis();
        System.out.println(Long.toHexString(crcValue));
        System.out.println((end - start) + " milliseconds");

        System.out.println("Mapped File:");
        start = System.currentTimeMillis();
        crcValue = checksumMappedFile(args[0]);
        end = System.currentTimeMillis();
        System.out.println(Long.toHexString(crcValue));
        System.out.println((end - start) + " milliseconds");
    }
}
```

java.io.FileInputStream 1.

- `FileChannel getChannel()` **1.4**

  Возвращает канал для доступа к потоку.

java.io.FileOutputStream 1.0

- `FileChannel getChannel()` **1.4**

  Возвращает канал для доступа к потоку.

java.io.RandomAccessFile 1.0

- `FileChannel getChannel()` **1.4**

  Возвращает канал для доступа к файлу.

java.nio.channels.FileChannel 1.

- `MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)`

  Отображает указанную область файла в память.

  Параметры

`mode`	Одна из констант, `READ_ONLY`, `READ_WRITE` или `PRIVATE`, определенных в классе `FileChannel.MapMode`
`position`	Начало отображаемой области
`size`	Размер отображаемой области

java.nio.Buffer 1.4

- `boolean hasRemaining()`

Возвращает значение `true`, если текущая позиция буфера еще не достигла граничного значения.

- `int limit()`

  Возвращает граничную позицию буфера, т.е. позицию, в которой данные уже недоступны.

API **java.nio.ByteBuffer 1.4**

- `byte get()`

  Возвращает байт, извлеченный из указанной позиции, и перемещает указатель на следующий байт.

- `byte get(int index)`

  Возвращает байт, извлеченный из позиции с указанным индексом.

- `ByteBuffer put(byte b)`

  Записывает байт в текущую позицию и перемещает указатель на следующий байт. Возвращает ссылку на буфер.

- `ByteBuffer put(int index, byte b)`

  Записывает байт в позицию с указанным индексом и перемещает указатель на следующий байт. Возвращает ссылку на буфер.

- `ByteBuffer get(byte[] destination)`
- `ByteBuffer get(byte[] destination, int offset, int length)`

  Заполняет массив или его часть байтами, считанными из буфера, и перемещает указатель вперед на число считанных байтов. Если в буфере осталось меньше байтов, чем требуется, то они не считываются и генерируется исключение `BufferUnderflowException`. Возвращает ссылку на буфер.

Параметры	`destination`	Заполняемый байтовый массив
	`offset`	Смещение заполняемой области
	`length`	Длина заполняемой области

- `ByteBuffer put(byte[] source)`
- `ByteBuffer put(byte[] source, int offset, int length)`

  Записывает все байты из массива или его части в буфер и перемещает указатель вперед, на число записанных байтов. Если в буфере осталось меньше места, чем требуется, то байты не записываются и генерируется исключение `BufferOverflowException`. Возвращает ссылку на буфер.

Параметры	`source`	Байтовый массив, предназначенный для записи
	`offset`	Смещение записываемой области
	`length`	Длина записываемой области

- *Xxx* `get`*Xxx*`()`
- *Xxx* `get`*Xxx*`(int index)`
- `ByteBuffer put`*Xxx*`(`*xxx*`value)`
- `ByteBuffer put`*Xxx*`(int index, `*xxx*`value)`

Эти методы используются для считывания и записи двоичных значений. Символы *Xxx* означают имена классов `Int`, `Long`, `Char`, `Float` или `Double`.

- `ByteBuffer order(ByteBuffer order)`
- `ByteBuffer order()`

  Задает или определяет порядок записи байтов. Значение параметра `order` равно одной из констант, `BIG_ENDIAN` или `LITTLE_ENDIAN`, определенных в классе `ByteOrder`.

Структура буфера

При отображении памяти создается буфер, который обслуживает весь файл либо ту его часть, которая интересует разработчика. Кроме того, при обмене данными используются буферы, которые хранят фрагменты данных.

В данном разделе мы вкратце рассмотрим основные операции, которые можно выполнить посредством объектов `Buffer`. Буфер представляет собой массив значений определенного типа. Класс `Buffer` является абстрактным; для создания конкретных объектов используются его подклассы `ByteBuffer`, `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer` и `ShortBuffer`. Наиболее часто применяются классы `ByteBuffer` и `CharBuffer`. Как видно на рис. 12.13, буфер имеет следующие характеристики.

- *Объем*, который никогда не изменяется.
- *Позиция* для чтения или записи значения.
- *Граница*, за пределами которой операции чтения и записи не имеют смысла.
- Необязательный *маркер* для повторного чтения или записи.

Эти характеристики удовлетворяют следующему соотношению:

$0 \le Маркер \le Позиция \le Граница \le Объем$

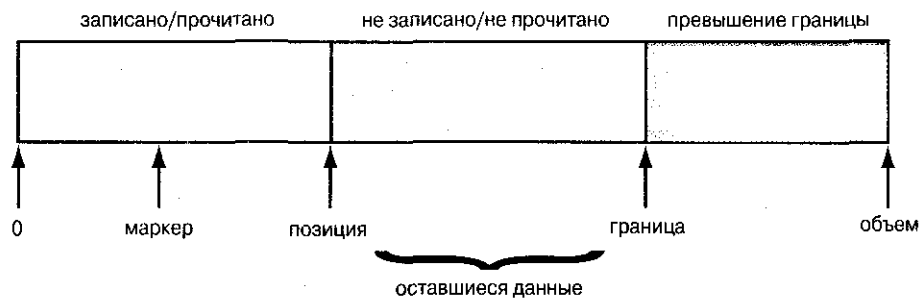


Рис. 12.13. Структура буфера

Буфер в основном предназначен для записи и чтения в цикле. В начале работы текущая позиция равна нулю, а граница совпадает с объемом буфера. Когда данные оканчиваются или объем буфера оказывается исчерпанным, пора переходить к чтению.

Для того чтобы установить границу равной текущей позиции, а позицию равной нулю, надо вызвать метод `flip()`. Метод `get()` можно вызывать до тех пор, пока значение, возвращаемое методом `remaining()` (граница минус позиция), положи-

тельно. Прочитав все данные из буфера, вызовите метод `clear()`, чтобы подготовить буфер к следующему циклу записи. В результате выполнения метода `clear()` позиция становится равной нулю, а граница – равной объему.

Если вы хотите повторно прочитать содержимое буфера, используйте `rewind()` или `mark()/reset()`. Более подробную информацию можно найти в документации по API.

java.nio.Buffer 1.4

- `Buffer clear()`

  Подготавливает буфер к записи, устанавливая позицию равной нулю, а границу – равной объему. Возвращает ссылку `this`.

- `Buffer flip()`

  Подготавливает буфер к чтению, устанавливая границу равной текущей позиции, а позицию – равной нулю. Возвращает ссылку `this`.

- `Buffer rewind()`

  Подготавливает буфер к повторному чтению тех же значений. позиция становится равной нулю, а граница не изменяется. Возвращает ссылку `this`.

- `Buffer mark()`

  Устанавливает маркер в текущую позицию. Возвращает ссылку `this`.

- `Buffer reset()`

  Устанавливает позицию равной маркеру, что позволяет повторно читать или записывать данные. Возвращает ссылку `this`.

- `int remaining()`

  Возвращает число значений, доступных для чтения или для записи. Другими словами, возвращаемое значение представляет собой разность между границей и позицией.

- `int position()`

  Возвращает позицию в буфере.

- `int capacity()`

  Возвращает объем буфера.

java.nio.CharBuffer 1.4

- `char get()`
- `CharBuffer get(char[] destination)`
- `CharBuffer get(char[] destination, int offset, int length)`

  Получает из буфера одно или несколько символьных значений, начиная с текущей позиции. Позиция сдвигается на количество прочитанных символов. Последние два метода возвращают ссылку `this`.

- `CharBuffer put(char c)`
- `CharBuffer put(char[] source)`

- `CharBuffer put(char[] source, int offset, int length)`
- `CharBuffer put(String source)`
- `CharBuffer put(CharBuffer source)`

  Помещает в буфер одно или несколько символьных значений, начиная с текущей позиции. Позиция сдвигается на количество записанных символов. Если в качестве параметра указан объект `CharBuffer`, читаются все оставшиеся символы. Все методы возвращают ссылку `this`.

- `CharBuffer read(CharBuffer destination)`

  Получает из буфера символьные значения и помещает их в объект `destination` до тех пор, пока не будет достигнута граница `destination`. Возвращает ссылку `this`.

Блокировка файлов

Рассмотрим ситуацию, в которой несколько программ, выполняемых одновременно, стремятся модифицировать один и тот же файл. Очевидно, что программы должны каким-то образом взаимодействовать, иначе файл легко повредить.

Для управления доступом к файлу или к его части можно использовать блокировку. Однако блокировка файлов сильно зависит от конкретной операционной системы. Этим объясняется тот факт, что средства блокировки файлов отсутствовали в ранних версиях пакета JDK.

Честно говоря, блокировка файлов нужна не во всех приложениях. Многие из них работают с базами данных, в которых есть свой механизм для решения проблемы одновременного доступа. Если информация хранится в обычном файле и вас беспокоит проблема одновременного обращения к нему, возможно, вам лучше применить базу данных и не вникать в сложные детали схем блокировки.

Однако на практике по-прежнему возникают ситуации, в которых без блокировки файлов не обойтись. Предположим, приложение сохраняет конфигурационные данные в файле. Если пользователь вызовет два экземпляра приложения, они могут попытаться одновременно записать информацию в один и тот же файл. В этой ситуации первый экземпляр должен заблокировать файл. Тогда второй экземпляр будет вынужден подождать, пока файл не окажется разблокированным, или прекратить попытки записи.

Чтобы заблокировать файл, нужно вызвать метод `lock()` или `tryLock()` класса `FileChannel`.

```
FileLock lock = channel.lock();
```

или

```
FileLock lock = channel.tryLock();
```

Если объект `lock` недоступен, первый метод приостанавливает свою работу и ожидает разблокирования файла. Второй вызов выполняется немедленно и либо блокирует объект, либо возвращает ссылку `null`, если объект `lock` недоступен.

Можно также заблокировать только часть файла:

```
FileLock lock(long start, long size, boolean exclusive)
```

или

```
FileLock tryLock(long start, long size, boolean exclusive)
```

Если файл блокируется для чтения и записи, параметр `exclusive` должен быть равен `true`. В этом случае говорят об *исключительной блокировке*. Если используется *совместная блокировка*, позволяющая нескольким процессам одновременно считывать данные из файла, в качестве параметра `exclusive` задается значение `false`. В результате ни один из процессов не может получить исключительные права доступа к файлу. Не все операционные системы поддерживают совместную блокировку. В некоторых случаях, запрашивая совместную блокировку, вы можете получить эксклюзивную. Для того чтобы определить, допускается ли совместная блокировка, следует вызвать метод `isShared()` класса `FileLock`.

Если последняя часть файла заблокирована, а сам файл увеличивается в размерах, дополнительная часть файла не блокируется. Чтобы заблокировать все байты, следует использовать константу `Long.MAX_VALUE`.

Учтите, что блокировка файла зависит от операционной системы. При разработке программ необходимо учитывать следующие особенности.

- В некоторых системах блокировка запрашивается, но не обязательно предоставляется. Если приложению не удается заблокировать файл, он остается доступным.
- В некоторых системах файл нельзя одновременно блокировать и отображать в память.
- Блокировка файлов поддерживается всей виртуальной машиной языка Java. Если две программы запускаются одной и той же виртуальной машиной, они не могут одновременно блокировать один и тот же файл. Если это произойдет, методы `lock()` и `tryLock()` сгенерируют исключение `OverlappingFileLockException`.
- В некоторых системах при закрытии канала файл, заблокированный виртуальной машиной Java, разблокируется. Следовательно, следует избегать использования нескольких каналов для одного и того же заблокированного файла.
- Блокирование файлов в сетевой файловой системе существенно зависит от операционной среды. По возможности блокирования таких файлов следует избегать.

`java.nio.channels.FileChannel` 1.4

- `FileLock lock()`

  Запрашивает исключительную блокировку всего файла. Метод приостанавливает свою работу до тех пор, пока не будут созданы условия для успешного завершения операции.

- `FileLock tryLock()`

  Запрашивает исключительную блокировку всего файла. Если файл заблокировать невозможно, возвращается ссылка `null`.

- `FileLock lock(long position, long size, boolean shared)`
- `FileLock tryLock(long position, long size, boolean shared)`

Эти методы запрашивают блокировку части файла. Если файл заблокировать невозможно, первый метод приостанавливает свое выполнение, а второй возвращает ссылку null.

Параметры	position	Начало блокируемого участка
	size	Размер блокируемого участка
	shared	Принимает значение true, если требуется совместная блокировка, и false — при исключительной блокировке

java.nio.channels.FileLock 1.4

- `void release()`

  Разблокирует файл.

Регулярные выражения

Регулярные выражения используются для определения строк, используемых в качестве шаблонов. Затем проверяется, совпадает ли строка с заданным шаблоном. Например, в одной из программ, которые будут рассмотрены далее в качестве примеров, производится поиск гипертекстовых ссылок, находящихся в HTML-файле, путем сравнения их с шаблоном `<a href="...">`.

Разумеется, в определении шаблона многоточие не задается. Нужно совершенно точно указать, какая последовательность символов считается допустимой. Для этого существуют определенные синтаксические правила.

В качестве примера рассмотрим следующее регулярное выражение:

```
[Jj]ava.+
```

Ему соответствует любая строка, удовлетворяющая правилам, перечисленным ниже.

- Первой буквой строки является `J` или `j`.
- Следующие три буквы — `ava`.
- Остальная часть строки может быть произвольной.

Например, строка `"javanese"` соответствует заданному регулярному выражению, а строка `"Core Java"` — нет.

Чтобы понимать смысл регулярных выражений, необходимо знать синтаксические правила, используемые для их построения. В большинстве случаев оказывается вполне достаточно нескольких синтаксических конструкций.

- *Класс символов* (character class) представляет собой набор символов, помещенный в квадратные скобки, например `[Jj]`, `[0-9]`, `[A-Za-z]` или `[^0-9]`. Для сравнения со строкой используется любой символ из набора. Здесь знак - означает диапазон (все символы Unicode, лежащие между двумя указанными символами), а ^ означает дополнение (любой символ, не отличный от указанного).
- Существуют предопределенные классы символов, например `\d` (цифры) или `\p{Sc}` (знак денежной единицы в системе Unicode). Подробно символы, применяемые для формирования регулярных выражений, описаны в табл. 12.8 и 12.9.

- Для большинства символов выполняется непосредственное сравнение. В качестве примера можно привести символы `ava` в рассмотренном выше регулярном выражении.
- Символ `.` соответствует любому символу (за исключением, возможно, символа конца строки; это зависит от значения флага).
- В качестве управляющего используется обратная косая черта, например, символы `\.` определяют точку, а пара символов `\\` — саму обратную косую черту.
- Символы `^` и `$` определяют соответственно начало и конец строки.
- *Если шаблоны* `X` *и* `Y` *являются регулярными выражениями, то шаблон* `XY` *означает* "любая строка, соответствующая выражению `X`, за которой следует строка, соответствующая выражению `Y`". Выражение `X|Y` означает "любая строка, соответствующая либо `X`, либо `Y`".
- К выражению `X` можно применять *итераторы*. `X+` означает одно или более вхождений выражения `X`, а `X*` — любое, в том числе нулевое число вхождений `X`, `X?` — необязательное вхождение выражения `X`.
- По умолчанию при использовании итераторов и символа `?` учитывается максимально возможное количество символов, гарантирующее соответствие строки с шаблоном. Это соглашение можно изменить, указав суффиксы `?` (учет наименьшего количества символов) и `+` (строгий учет наибольшего количества символов, даже если при этом нарушается соответствие всей строки шаблону).

  Например, строка `cab` удовлетворяет шаблону `[a-z]*ab`, но не шаблону `[a-z]*+ab`. В первом случае выражению `[a-z]*` соответствует только символ `c`, так что символы `ab` соответствуют оставшейся части шаблона. Однако строгий учет *максимального количества символов ставит в соответствие выражению* `[a-z]*` всю строку `cab`, несмотря на то, что символов для оставшейся части шаблона уже не остается.
- Фрагменты регулярных выражений можно оформлять как *группы*. Группы помещаются в скобки `()`, например `([+-]?)([0-9]+)`. Группы нумеруются начиная с единицы и для указания номера группы используются выражения `\n`, где `n` — число.

Рассмотрим довольно сложное, но полезное регулярное выражение. Оно описывает десятичные или шестнадцатеричные целые числа.

```
[+-]?[0-9]+|0[Xx][0-9A-Fa-f]+
```

К сожалению, синтаксис выражения еще не полностью стандартизован и варьируется в разных программах и библиотеках. В настоящее время достигнуто общее соглашение об основных конструкциях, но осталось много неясностей в деталях. Классы регулярных выражений используют синтаксис, очень похожий на синтаксис языка Perl, *хотя и не совпадающий с ним. Все эти синтаксические конструкции приведены* в табл. 12.4. Более подробно регулярные выражения описаны в документации по API (класс `Pattern`). Читателям, стремящимся глубже изучить этот вопрос, мы можем порекомендовать книгу Джеффри Е. Ф. Фридла (Jeffrey E. F. Friedl) *Mastering Regular Expressions* (O'Reilly and Associates, 1997).

Простейший способ применения регулярных выражений — проверка соответствия ему конкретной строки. На языке Java это выглядит следующим образом. Сначала с ис-

пользованием строки, задающей регулярное выражение, создается объект `Pattern`. Затем с помощью объекта `Pattern` формируется объект `Master` и вызывается метод `matches()`.

```
Pattern patter = Pattern.compile(patternString);
Matcher matcher = pattern.matcher(input);
if(matcher.matches()) ...
```

Таблица 12.8. Символы, используемые для формирования регулярных выражений

Символ	Описание
`c`	Символ `c`
`\unnnn`, `\xnn`, `\0n`, `\0nn`, `\0nnn`	Символ, имеющий заданный десятичный или шестнадцатеричный код
`\t`, `\n`, `\r`, `\f`, `\a`, `\e`	Символы табуляции, перевода строки, возврата каретки, перевода формата, сигнала и замены символа `esc`
`\c`*c*	Управляющий символ, соответствующий символу *c*
Классы символов	
`[`C_1C_2`...]`	Любой символ, представленный в списке C_1C_2... Знак C_i означает символ, диапазон (C_1–C_2) или класс символов
`[^...]`	Дополнение класса символов
`[...&&...]`	Пересечение двух классов символов
Предопределенные классы символов	
`.`	Любой символ, за исключением символа конца строки (или любой символ, если установлен флаг `DOTALL`)
`\d`	Цифра `[0-9]`
`\D`	Не цифра `[^0-9]`
`\s`	Один из символов `[\t\n\r\f\x0B]`
`\S`	Символ, не совпадающий с `[\t\n\r\f\x0B]`
`\w`	Буква или цифра
`\W`	Не буква и не цифра
`\p{`*имя*`}`	Именованный класс символов (см. табл. 12.5)
`\P{`*имя*`}`	Дополнение именованного класса символов
Ограничители	
`^$`	Начало и конец ввода (или начало и конец строки в многострочном режиме)
`\b`	Граница слова
`\B`	Граница не слова
`\A`	Начало ввода
`\z`	Конец ввода
`\Z`	Конец ввода, за исключением признака конца последней строки
`\G`	Конец последовательности, соответствующей предыдущей части выражения

Окончание табл. 12.8

Символ	Описание
Итераторы	
`X?`	Необязательная строка, соответствующая `X`
`X*`	Любое, в том числе нулевое, число повторений строки, соответствующей `X`
`X+`	Одно и более повторений строки, соответствующей `X`
`X{n}`, `X{n,}`, `X{n, m}`	Строка, соответствующая `X`, повторенная *n* раз, не менее чем *n* раз, от *n* до *m* раз
Суффиксы итераторов	
`?`	Учет минимального количества повторений символов
`+`	*Строгий учет максимального количества повторений* символов
Сложные выражения	
`XY`	Любая строка, начинающаяся строкой, соответствующей `X`, за которой следует строка, соответствующая `Y`
`X|Y`	Строка, соответствующая либо `X`, либо `Y`
Группировка	
`(X)`	Объявляет X как группу
`\n`	Соответствие *n*-й группе
Управляющие символы	
`\c`	*Символ c (не должен быть обычным символом)*
`(?...)`	Специальная конструкция (см. описание класса `Pattern` в документации по API)

Таблица 12.9. Имена предопределенных классов символов

`Lower`	*Строчные ASCII-символы* `[a-z]`
`Upper`	Прописные ASCII-символы `[A-Z]`
`Alpha`	Буквы в кодировке ASCII `[A-Za-z]`
`Digit`	Цифры в кодировке ASCII `[0-9]`
`Alnum`	Буквы или цифры в кодировке ASCII `[A-Za-z0-9]`
`XDigit`	Шестнадцатеричные цифры `[0-9A-Fa-f]`
`Print` или `Graph`	Отображаемые символы в кодировке ASCII `[\x21-\x7E]`
`Punct`	Не буквы и не цифры в кодировке ASCII `[\p{Print}&&\P{Alnum}]`
`ASCII`	Все символы в кодировке ASCII `[\x00-\x7F]`
`Cntrl`	*Управляющие символы в кодировке ASCII* `[\p{ASCII}&&\P{Print}]`
`Blank`	Пробел или символ табуляции `[\t]`
`Space`	Один из символов `[\t\n\r\f\x0B]`

Окончание табл. 12.9

`javaLowerCase`	Нижний регистр символов, определяемый `Character.isLowerCase()`
`javaUpperCase`	Верхний регистр символов, определяемый `Character.isUpperCase()`
`javaWhitespace`	Пробел, определяемый `Character.isWhitespace()`
`javaMirrored`	Способность к отражению, определяемая `Character.isMirrored()`
`In`*Блок*	*Блок* — это имя блока Unicode-символов, из которого удален пробел, например `BasicLatin` или `Mongolian`. Список имен блоков приведен по адресу `http://www.inicode.org`
Категория или `In`*Категория*	*Категория* — это имя категории Unicode-символов, например `L` (буква) или `Sc` (символ денежной единицы). Список имен категорий приведен по адресу `http://www.inicode.org`

В качестве параметра методу `matcher()` передается объект любого класса, реализующего интерфейс `CharSequence`, например `String`, `StringBuffer` или `CharBuffer`.

При компиляции шаблона можно установить один или несколько флагов. Например:

```
Pattern pattern = Pattern.compile)patternString,
   Pattern.CASE_INSENSITIVE + Pattern.CASE_UNICODE_CASE);
```

В настоящее время поддерживаются шесть флагов.

- `CASE_INSENSITIVE`. Сравнение символов без учета регистра (строчные и прописные буквы не различаются). По умолчанию этот флаг учитывает лишь символы из набора US ASCII.
- `UNICODE_CASE`. В сочетании с флагом `CASE_INSENSITIVE` задает учет регистра, распознаваемого средствами Unicode.
- `MULTILINE`. Символы `^` и `$` означают начало и конец строки, но не всей символьной последовательности.
- `UNIX_LINE`. Признаком окончания строки считается только символ `'\n'`.
- `DOTALL`. Если установлен этот флаг, символ `.` совпадает с любым символом, включая символ конца строки.
- `CANON_EQ`. Учитывает каноническую эквивалентность символов в системе Unicode. Например, символ u, за которым следует¨, соответствует букве ü.

Если регулярное выражение содержит группы, то объект `Matcher` может определить их границы. Приведенные ниже методы определяют соответственно индекс начала группы и индекс, следующий за ее окончанием.

```
Int start(int groupIndex)
int end(inr groupIndex)
```

Строку, соответствующую группе, можно выделить с помощью следующего вызова:

```
String group(int groupIndex)
```

Группа 0 означает всю входную строку, индекс первой реальной группы равен 1. Для подсчета групп следует вызывать метод `groupCount()`.

Нумерация вложенных групп осуществляется с учетом скобок. Рассмотрим шаблон

```
(1(1?[0-9]):([0-5][0-9]))[ap]m
```

Допустим, что мы сравниваем с ним следующую строку:

```
11:59am
```

Объект `Matcher` сообщит следующее.

Индекс группы	Начало	Конец	Строка
0	0	7	11:59am
1	0	5	11:59
2	0	2	11
3	3	5	59

Программа, код которой приведен в листинге 12.9, предлагает ввести шаблон, а затем строку для сравнения. В результате работы программа определяет, соответствует ли строка заданному шаблону. Если результат сравнения положителен и при этом шаблон содержит группы, программы отмечает границы групп с помощью скобоки. Например:

```
((11):(59))am
```

Листинг 12.9. Содержимое файла `RegexTest.java`

```
import java.util.*;
import java.util.regex.*;

/**
   Данная программа проверяет соответствие строки
   регулярному выражению. Введите шаблон и строку,
   подлежащую сравнению, либо щелкните на кнопке Cancel,
   чтобы завершить работу программы. Если шаблон содержит
   группы, их границы выделяются скобками.
*/
public class RegExTest
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.println("Enter pattern: ");
      String patternString = in.nextLine();

      Pattern pattern = null;
      try
      {
         pattern = Pattern.compile(patternString);
      }
      catch (PatternSyntaxException e)
      {
         System.out.println("Pattern syntax error");
```

```
            System.exit(1);
         }

         while (true)
         {
            System.out.println("Enter string to match: ");
            String input = in.nextLine();
            if (input == null || input.equals("")) return;
            Matcher matcher = pattern.matcher(input);
            if (matcher.matches())
            {
               System.out.println("Match");
               int g = matcher.groupCount();
               if (g > 0)
               {
                  for (int i = 0; i < input.length(); i++)
                  {
                     for (int j = 1; j <= g; j++)
                        if (i == matcher.start(j))
                           System.out.print('(');
                     System.out.print(input.charAt(i));
                     for (int j = 1; j <= g; j++)
                        if (i + 1 == matcher.end(j))
                           System.out.print(')');
                  }
                  System.out.println();
               }
            }
            else
               System.out.println("No match");
         }
      }
   }
```

Часто с регулярным выражением сравнивается не вся входная строка, а лишь ее часть. Для определения очередного совпадения следует применять метод `find()` класса `Matcher`. Если он возвращает значение `true`, используйте методы `start` и `end` для определения границ последовательности символов, выдержавшей сравнение.

```
while (mathcer.find())
{
   int start = matcher.start();
   int end = matcher.end();
   String match = input.substring(start, end);
}
```

Программа, исходный текст которой показан в листинге 12.10, демонстрирует работу этого механизма. Она распознает все гипертекстовые ссылки, расположенные на Web-странице, и выводит их на экран. При запуске программы укажите в командной строке URL.

```
java HrefMatch http://www.horstmann.com
```

Листинг 12.10. Содержимое файла `HrefMatch.java`

```
import java.io.*;
import java.net.*;
import java.util.regex.*;

/**
   Программа выводит все URL, содержащиеся на Web-странице,
   сравнивая код документа с регулярным выражением,
   описывающим HTML-дескриптор <a href=...>.
   Для запуска программы следует выполнить команду
   java HrefMatch URL
*/
public class HrefMatch
{
   public static void main(String[] args)
   {
      try
      {
         // Получение URL из командной строки или
         // использование значения по умолчанию.
         String urlString;
         if (args.length > 0) urlString = args[0];
         else urlString = "http://java.sun.com";

         // Создание потока, связанного с URL.
         InputStreamReader in = new InputStreamReader(new
            URL(urlString).openStream());

         // Чтение данных в буфер.
         StringBuilder input = new StringBuilder();
         int ch;
         while ((ch = in.read()) != -1) input.append((char) ch);

         // Поиск всех вхождений последовательности символов,
         // соответствующей шаблону.
         String patternString =
            "<a\\s+href\\s*=\\s*(\"[^\"]*\"|[^\\s>])\\s*>";
         Pattern pattern = Pattern.compile(patternString,
            Pattern.CASE_INSENSITIVE);
         Matcher matcher = pattern.matcher(input);

         while (matcher.find())
         {
            int start = matcher.start();
            int end = matcher.end();
            String match = input.substring(start, end);
            System.out.println(match);
         }
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
      catch (PatternSyntaxException e)
      {
```

```
            e.printStackTrace();
        }
    }
}
```

Метод `replaceAll()` класса `Matcher` заменяет все элементы регулярного выражения соответствующей строкой. Например, приведенный ниже фрагмент программы заменяет все последовательности цифр символом #.

```
Pattern pattern = Pattern.compile("[0-9]+");
Mathcer mathcer = pattern.matcher(input);
String output = mathcer.replaceAll("#");
```

Заменяющая строка может содержать ссылки на группы элементов шаблона: символ `$n` заменяется *n*-й группой. Для включения символа `$` в заменяющий текст нужно использовать обозначение `\$`.

Существует также метод `replaceFirst()`, заменяющий первое вхождение элемента шаблона.

И наконец, класс `Pattern` содержит метод `split()`, который разбивает входную строку на массив строк, используя в качестве границ элементы строки, совпадающие с регулярным выражением. Например, следующий фрагмент кода разбивает строку на лексемы, причем в качестве разделителей используются знаки пунктуации, окруженные необязательными пробелами.

```
Pattern pattern = Pattern.compile("\\s*\\p{Punct}\\s*");
String[] tokens = pattern.split(input);
```

API **java.util.regex.Pattern 1.4**

- `static Pattern compile(String expression)`
- `static Pattern compile(String expression, int flag)`

  Компилирует регулярное выражение в объект шаблона. Это позволяет быстро выполнять сравнение.

Параметры	`expression`	Регулярное выражение
	`flag`	Один или несколько флагов, задаваемых константами `CASE_INSENSITIVE`, `UNICODE_CASE`, `MULTILINE`, `UNIX_LINES`, `DOTALL` и `CANON_EQ`

- `Matcher matcher(CharSequence input)`

  Возвращает объект класса `Matcher`, который можно использовать для сравнения входной строки с шаблоном.

- `String[] split(CharSequence input)`
- `String[] split(CharSequence input, int limit)`

  Разбивает входную строку на лексемы, используя разделители, заданные посредством шаблона. Возвращает массив лексем. Разделители в состав лексем не входят.

Параметры	`input`	Строка, разделяемая на лексемы
	`limit`	Максимальное количество строк, полученных в результате. Если распознано `limit-1` разделителей,

то последним элементом возвращаемого массива является оставшаяся часть строки, не разделенная на лексемы. Если `limit≤0`, то на лексемы разбивается вся входная строка. Если `limit=0`, то оставшиеся пустые строки в массив не записываются

java.util.regex.Matcher

- `boolean matches()`

  Если входная строка соответствует шаблону, возвращает значение `true`.

- `boolean lookingAt()`

  Если начало входной строки соответствует шаблону, возвращает значение `true`.

- `boolean find()`
- `boolean find(int start)`

  Пытается найти очередную последовательность символов, соответствующую регулярному выражению, и в случае удачи возвращает значение `true`.

  Параметр `start` Индекс, с которого начинается поиск

- `int start()`
- `int end()`

  Возвращает индекс начала последовательности символов, соответствующей шаблону, и индекс первого символа, не входящего в последовательность.

- `String group()`

  Возвращает текущий фрагмент, соответствующий регулярному выражению.

- `int groupCount()`

  Возвращает количество групп в шаблоне.

- `int start(int groupIndex)`
- `int end(int groupIndex)`

  Возвращает первый индекс последовательности символов, соответствующей группе, и индекс первого символа, не входящего в последовательность.

  Параметр `groupIndex` Индекс группы (начиная с 1) или 0 в случае соответствия всему шаблону

- `String group(int groupIndex)`

  Возвращает текущее соответствие.

  Параметр `groupIndex` Индекс группы (начиная с 1) или 0 в случае соответствия всему шаблону

- `String replaceAll(String replacement)`
- `String replaceFirst(String replacement)`

  Возвращает строку, полученную из входной строки путем замены строкой `replacement` либо всех последовательностей символов, соответствующих шаблону, либо первой последовательности.

Параметр	`replacement`	Заменяющая строка. Может содержать ссылки `$`*n* на группы шаблона. Для включения символа `$` применяется обозначение `\$`

- `Matcher reset()`
- `Matcher reset(CharSequence input)`

  Обновляет состояние объекта класса `Matcher`. Второй метод выполняет переход на другую входную строку. Оба метода возвращают указатель `this`.

Глава 13

Работа с универсальными типами

В этой главе...

- Зачем нужны универсальные типы
- Создание простого универсального класса
- Универсальные методы
- Ограничения, накладываемые на переменные типа
- Универсальный код и виртуальная машина Java
- Ограничения на работу с универсальными типами
- Правила наследования для универсальных типов
- Подстановочные типы
- Отражение и универсальные типы

Введение универсальных типов — одно из наиболее существенных изменений языка Java, начиная с версии 1.0. Несмотря на то что универсальные типы были реализованы только в JDK 5.0, подобные требования были сформулированы в документе Java Specification Requests (JSR) 14 еще в 1999 году. Группе специалистов пришлось потратить около пяти лет на составление спецификации и тестирование новых средств.

Универсальные типы позволяют создавать более надежный и удобный для восприятия код, в котором удается сократить число переменных `Object` и исключить ненужные операции приведения типов. Новые средства очень полезны при работе с наборами данных, в частности с широко применяемыми списочными массивами.

Универсальные типы можно сравнить с шаблонами, используемыми в языке C++. Шаблоны были добавлены в C++ для поддержки типизированных наборов данных. Однако через длительное время выяснилось, что шаблоны имеют более широкое применение. Аналогично обстоит дело и с универсальными типами Java. При чтении этой главы обратите внимание, как именно можно применить эти типы при написании программ.

Зачем нужны универсальные типы

Универсальные типы позволяют создавать код, который может быть повторно использован для работы с различными объектами. Вряд ли вы захотите создавать отдельные классы для наборов строк и объектов `File`. Это и не нужно. Класс `ArrayList` позволяет хранить объекты любого типа. Это лишь один из примеров применения универсальных типов.

До появления JDK 5.0 универсальные типы можно было реализовать только посредством наследования. Класс `ArrayList` может содержать лишь ссылки на объекты `Object`.

```
public class ArrayList // До появления JDK 5.0
{
   public Object get(int i) { . . . }
   public void add(Object o) { . . . }
   . . .
   private Object[] elementData;
}
```

При таком подходе возникают две проблемы. Во-первых, для того, чтобы получить требуемые данные, необходимо выполнять приведение типов.

```
ArrayList files = new ArrayList();
. . .
String filename = (String) names.get(0);
```

Во-вторых, невозможно выполнить проверку на наличие ошибок. Списочный массив позволяет включать в него любые объекты.

```
files.add(new File(". . ."));
```

При компиляции и выполнении такого выражения ошибка не возникает. Она проявляется лишь при попытке преобразовать `Object` в объект `String`.

В JDK 5.0 было предложено решение этих проблем. Оно состоит в использовании *параметров типа*. Теперь для класса `ArrayList` можно явно указать тип содержащихся в нем элементов.

```
ArrayList<String> files = new ArrayList<String>();
```

Это упрощает восприятие кода, поскольку в программе явно указано, что списочный массив может содержать только объекты `String`.

Данную дополнительную информацию использует и компилятор. При вызове метода `get()` приведение типов не требуется; компилятор знает, что возвращается объект `String`, а не `Object`.

```
String filename = files.get(0);
```

Компилятор также знает, что методу `add()` объекта `ArrayList<String>` должен быть передан параметр `String`. Это гораздо надежнее, чем использование параметра `Object`. Компилятор может проверить, не включаете ли вы объект, тип которого не соответствует указанному. Так, например, приведенное ниже выражение не будет скомпилировано.

```
files.add(new File(". . ."); // В ArrayList<String> можно
                             // включать только объекты String
```

Гораздо лучше получить сообщение об ошибке на этапе компиляции, чем выяснять причины возникновения исключения при выполнении программы.

Сказанное выше объясняет причину, по которой параметры типа так привлекательны для разработчиков. Благодаря их наличию код легче воспринимается и программа работает более надежно.

Кто создает универсальные типы

Использовать универсальные классы типа `ArrayList` несложно. Большинство программистов, создающих Java-приложения, без труда включат в свою программу класс, подобный `ArrayList<String>`, при условии, что он является встроенным средством языка, как, например, массив `String[]`. (Очевидно, что списочный массив удобнее обычного, поскольку он может автоматически расширяться в процессе выполнения программы.)

Однако реализовать универсальный класс непросто. Разработчикам приходится указывать в качестве параметров типа любые классы. Полученные конструкции должны работать без ограничений и неожиданных ошибок. Поэтому при создании универсального класса надо предусмотреть все возможные варианты его применения.

Сложно ли это? Перед разработчиками библиотеки стандартных классов встает типичная проблема. Класс `ArrayList` содержит метод `addAll()`, предназначенный для включения всех элементов другого набора. Разработчику может понадобиться включить в `ArrayList<Employee>` все элементы из `ArrayList<Manager>`. Возникает задача: разрешить вызовы одного вида и запретить другие. Разработчики языка Java нашли удачное решение этой задачи, введя понятие *подстановочного типа* (wildcard type). Подстановочный тип — это абстракция, но она позволяет разработчикам библиотек реализовывать максимально гибкие методы.

Среди программистов, использующих и создающих универсальные типы, можно выделить три категории, соответствующие различным уровням подготовки. К первой категории относятся разработчики, которые лишь используют универсальные типы — обычно это наборы данных вроде `ArrayList` — и не задумываются о том, как они работают. Часто такой подход объясняется низким профессиональным уровнем. Большинство программистов, создающих прикладные программы, поступают именно так до тех пор, пока перед ними не возникнут проблемы, заставляющие несколько глубже познакомиться с механизмом универсальных типов. Обычно такой проблемой бывает непонятная ошибка, возникающая вследствие неправильного использования универсальных классов, либо необходимость взаимодействия с существующим кодом, в котором не поддерживаются параметры типов. В этом случае бессистемные попытки исправить ситуацию приводят лишь к дополнительным затратам времени. Детально изучив универсальные типы, можно гораздо быстрее справиться с проблемой. Разработчики, поступающие так, автоматически переходят во вторую категорию. И наконец, в процессе работы может понадобиться реализовать собственные универсальные классы и методы. Это задача для специалистов третьей категории, отличающихся высоким уровнем подготовки.

Разработчики прикладных программ редко создают большой объем кода, связанного с универсальными типами. Основную работу уже провели специалисты Sun и реализовали параметры типов для всех классов, описывающих наборы данных. На практике использование параметров типов оправдано лишь в тех программах, в которых очень часто используется приведение типов для низкоуровневых объектов (например, `Object` или `Comparable`).

В этой главе мы расскажем вам все, что необходимо знать для того, чтобы реализовать собственный универсальный класс. Однако мы считаем, что большинство читателей используют этот материал лишь для разрешения возникающих проблем и удовлетворения собственного любопытства о том, как работают классы, представляющие наборы данных.

Создание простого универсального класса

Универсальный класс — это класс, допускающий указание одной или нескольких переменных типа. В этой главе мы используем в качестве примера класс `Pair`. Поскольку класс очень прост, мы можем сосредоточить внимание на вопросах универсальности, не отвлекаясь на детали хранения данных. Ниже приведен код класса `Pair`.

```
public class Pair<T>
{
   public Pair() { first = null; second = null; }
   public Pair(T first, T second) { this.first = first;
      this.second = second; }

   public T getFirst() { return first; }
   public T getSecond() { return second; }

   public void setFirst(T newValue) { first = newValue; }
   public void setSecond(T newValue) { second = newValue; }

   private T first;
   private T second;
}
```

В классе `Pair` задается переменная типа `T`, которая располагается после имени класса и помещается в угловые скобки (< >). В универсальном классе может быть несколько переменных типа. Например, мы можем задать различные типы для первого и второго поля.

```
public class Pair<T, U> { . . . }
```

Переменные типы используются в коде класса, могут определять тип значения, возвращаемого методом, а также типы полей и локальных переменных. Пример указания типа поля приведен ниже.

```
private T first; // Используется переменная типа
```

Для переменных типа принято задавать короткие имена и формировать их из символов верхнего регистра. В библиотеке Java при определении наборов данных для указания типа элемента используется переменная E, для типов ключа и значения — соответственно имена K и V. Буквой T, а по мере необходимости также буквами U и S, обозначается произвольный тип.

При создании экземпляра универсального типа осуществляется подстановка переменных типа, например, так, как показано ниже.

```
Pair<String>
```

В результате получается обычный класс с конструкторами и методами. Их сигнатуры выглядят следующим образом.

Конструкторы:

```
Pair<String>()
Pair<String>(String, String)
```

Методы:

```
String getFirst()
String getSecond()
void setFirst(String)
void setSecond(String)
```

Другими словами, универсальный класс выступает в роли фабрики для обычных классов.

Программа, код которой приведен на рис. 13.1, демонстрирует использование класса `Pair`. Статический метод `minmax()` просматривает массив, одновременно определяя минимальный и максимальный из содержащихся в нем элементов. Оба значения возвращаются с помощью объекта `Pair`. Как вы, наверное, помните, метод `compareTo()` сравнивает две строки и возвращает 0, если они идентичны, отрицательное значение, если первая строка лексикографически предшествует второй, и положительное значение в противном случае.

Универсальные классы Java очень похожи на шаблоны C++. Единственное очевидное отличие в том, что в Java не используется специальное ключевое слово `template`. Однако, как вы узнаете позже в этой главе, различия между универсальными классами и шаблонами этим не исчерпываются.

Листинг 13.1. Содержимое файла `PairTest1.java`

```
public class PairTest1
{
   public static void main(String[] args)
   {
      String[] words = { "Mary", "had", "a", "little", "lamb" };
      Pair<String> mm = ArrayAlg.minmax(words);
      System.out.println("min = " + mm.getFirst());
      System.out.println("max = " + mm.getSecond());
   }
}

class ArrayAlg
{
   /**
      Находит минимальное и максимальное значение
      в массиве строк.
      @param a Массив строк
      @return Объект Pair, содержащий и минимальное занчение.
          Если a представляет собой пустой массив, возвращается
          значение null
   */
   public static Pair<String> minmax(String[] a)
   {
      if (a == null || a.length == 0) return null;
      String min = a[0];
```

```
      String max = a[0];
      for (int i = 1; i < a.length; i++)
      {
         if (min.compareTo(a[i]) > 0) min = a[i];
         if (max.compareTo(a[i]) < 0) max = a[i];
      }
      return new Pair<String>(min, max);
   }
}
```

Универсальные методы

В предыдущем разделе мы показали вам, как создается универсальный класс. При необходимости вы также можете определить метод с параметром типа.

```
class ArrayAlg
{
   public static <T> T getMiddle(T[] a)
   {
      return a[a.length / 2]);
   }
}
```

Этот метод не принадлежит универсальному классу; он определяется в составе обычного класса. Однако сам метод является универсальным; на это указывают угловые скобки и переменная типа. Обратите внимание на то, что переменные типа указываются после модификаторов (в нашем случае это `public static`) и перед типом возвращаемого значения.

Универсальные методы можно определять в составе как обычных, так и универсальных классов.

При вызове универсального метода перед его именем указываются реальные типы, помещенные в угловые скобки.

```
String[] names = { "John", "Q.", "Public" };
String middle = ArrayAlg.<String>getMiddle(names);
```

В данном случае (как, впрочем, и в подавляющем большинстве ситуаций) можно при вызове метода не указывать тип параметра `<String>`. Компилятор имеет достаточно информации для того, чтобы восстановить требуемую информацию. Он сравнивает тип переменной `names` (`String[]`) с универсальным типом `T[]` и делает вывод, что `T` соответствует `String`. Таким образом, вызов метода может выглядеть следующим образом:

```
String middle = ArrayAlg.getMiddle(names);
```

В C++ параметр типа указывается после имени метода. Это приводит к неоднозначности при разборе. Например, `g(f<a,b>(c))` может означать "вызов `g`, которому передается результат выполнения `f<a,b>(c)`", либо "вызов `g`, которому передаются два логических значения — `f<a` и `b>(c)`".

Ограничения, накладываемые на переменные типа

Иногда бывает необходимо ограничить переменные определенными типами для класса или метода. Рассмотрим в качестве примера задачу вычисления наименьшего элемента массива.

```
class ArrayAlg
{
   public static <T> T min(T[] a) // "Почти корректно"
   {
      if (a == null || a.length == 0) return null;
      T smallest = a[0];
      for (int i = 1; i < a.length; i++)
         if (smallest.compareTo(a[i]) > 0) smallest = a[i];
      return smallest;
   }
}
```

Однако в данном случае возникает проблема. Обратите внимание на код метода `min()`. Переменная `smallest` имеет тип `T`, а это значит, что она может ссылаться на любой класс. Откуда мы узнаем, что класс, которому соответствует `T`, содержит метод `compareTo()`?

Выход — ограничить `T` классами, реализующими интерфейс `Comparable`, в котором объявлен метод `compareTo()`. Ограничение на переменную `T` накладывается следующим образом:

```
public static <T extends Comparable> T min(T[] a) . . .
```

Интерфейс `Comparable` сам является универсальным типом, но на данном этапе мы игнорируем эту особенность.

Теперь универсальному методу `min()` может быть передан только массив, элементы которого реализуют интерфейс `Comparable`, например `String`, `Date` и т.д. Если методу `min()` будет передан массив объектов `Rectangle`, на этапе компиляции возникнет ошибка, поскольку класс `Rectangle` не реализует интерфейс `Comparable`.

В C++ ограничить типы параметров шаблонов невозможно. Если разработчик создает экземпляр неподходящего типа, используя шаблон, возникает ошибка, сообщение о которой носит весьма туманный характер.

Возможно, вы удивитесь, почему при ограничении типов указывается ключевое слово `extends`, а не `implements`, ведь `Comparable` — это интерфейс. Приведенная ниже запись сообщает о том, что `T` — это подтип ограничивающего типа.

```
<T extends Ограничивающий_Тип>
```

И `T`, и ограничивающий тип могут быть как классами, так и интерфейсами. Ключевое слово `extends` было выбрано, во-первых, из-за того, что оно явно отражает понятие подтипа, а во-вторых, потому, что разработчики Java не захотели вводить в состав языка новое ключевое слово (например, `sub`).

На переменную типа может быть наложено несколько ограничений. Например:

```
T extends Comparable & Serializable
```

Ограничивающие типы разделяются символом `&`. Запятую использовать для этой цели нельзя, поскольку она разделяет переменные типа.

Задавать можно сколько угодно супертипов, но, учитывая механизм наследования Java, классом может быть не более чем один из них. Если в качестве границы применяется класс, он должен быть указан первым в списке ограничивающих типов.

В листинге 13.2 мы переписали метод `minmax()` так, что он стал универсальным. Этот метод вычисляет минимальное и максимальное значения универсального массива, возвращая объект `Pair<T>`.

Листинг 13.2. Содержимое файла `PairTest2.java`

```
import java.util.*;

public class PairTest2
{
   public static void main(String[] args)
   {
      GregorianCalendar[] birthdays =
         {
            new GregorianCalendar(1906,
               Calendar.DECEMBER, 9), // Дж. Хоппер
            new GregorianCalendar(1815,
               Calendar.DECEMBER, 10), // А. Лавлейс
            new GregorianCalendar(1903,
               Calendar.DECEMBER, 3), // Дж. фон Нейман
            new GregorianCalendar(1910,
               Calendar.JUNE, 22), // К. Зус
         };
      Pair<GregorianCalendar> mm = ArrayAlg.minmax(birthdays);
      System.out.println("min = " + mm.getFirst().getTime());
      System.out.println("max = " + mm.getSecond().getTime());
   }
}

class ArrayAlg
{
   /**
      Определяет минимальное и максимальное значения
      объектов типа T.
      @param a Массив объектов типа T
      @return Объект Pair, содержащий минимальное и максимальное
         значения, либо ссылку null, если a представляет собой
         значение null, либо пустой массив
   */
   public static <T extends Comparable> Pair<T> minmax(T[] a)
   {
      if (a == null || a.length == 0) return null;
      T min = a[0];
      T max = a[0];
      for (int i = 1; i < a.length; i++)
      {
         if (min.compareTo(a[i]) > 0) min = a[i];
         if (max.compareTo(a[i]) < 0) max = a[i];
      }
      return new Pair<T>(min, max);
   }
}
```

Универсальный код и виртуальная машина Java

Виртуальная машина не поддерживает объектов универсальных типов. Все объекты, с которыми она работает, соответствуют обычным классам. Ранние реализации универсальных классов даже обеспечивали компиляцию программ в файлы классов, способных выполняться под управлением виртуальной машины версии 1.0. От подобной обратной совместимости пришлось позднее отказаться. В настоящее время при компиляции кода с универсальными типами с помощью компилятора Sun полученные файлы классов могут обрабатываться виртуальной машиной версии не ниже 5.0.

Если вы хотите воспользоваться преимуществами JDK 5.0 и в то же время сохранить совместимость с более старыми версиями виртуальных машин, воспользуйтесь программой Retroweaver (`http://sourceforge.net/projects/retroweaver`). Эта программа преобразует файлы классов для обработки ранними версиями виртуальных машин.

Если вы определите универсальный тип, он будет автоматически преобразован в соответствующий ему низкоуровневый код. Имя низкоуровневого типа создается на базе имени универсального типа, а параметры типа удаляются. Переменные типа заменяются ограничивающими типами; если ограничивающий тип не указан, используется тип `Object`.

Например, низкоуровневый тип, соответствующий `Pair<T>`, выглядит следующим образом:

```
public class Pair
{
   public Pair(Object first, Object second)
   {
      this.first = first;
      this.second = second;
   }

   public Object getFirst() { return first; }
   public Object getSecond() { return second; }

   public void setFirst(Object newValue) { first = newValue; }
   public void setSecond(Object newValue) { second = newValue; }

   private Object first;
   private Object second;
}
```

Поскольку тип `T` не ограничен, он заменяется типом `Object`.

В результате преобразования создается обычный класс, такой, какой был бы создан программистом до появления в языке Java универсальных типов.

В программе могут присутствовать различные разновидности класса `Pair`, например `Pair<String>` или `Pair<GregorianCalendar>`. Все они преобразуются в низкоуровневый класс `Pair`.

В этом отношении универсальные типы Java существенно отличаются от шаблонов C++. В языке C++ при создании каждого экземпляра, соответствующего шаблону, создаются различные типы. Это явление принято называть "разрастанием кода шаблонов". В Java подобные проблемы не возникают.

При замене переменных типа низкоуровневыми типами используется первое ограничение. Если ограничивающие типы не указаны, принимается тип `Object`. Например, для переменной типа в классе `Pair<T>` ограничения не заданы, поэтому `T` заменяется на `Object`. Предположим, что мы объявили несколько иной тип.

```
public class Interval<T extends Comparable & Serializable> implements
Serializable
{
   public Interval(T first, T second)
   {
      if (first.compareTo(second) <= 0) { lower = first; upper = second; }
      else { lower = second; upper = first; }
   }
   . . .
   private T lower;
   private T upper;
}
```

Низкоуровневый код для `Interval` будет выглядеть следующим образом.

```
public class Interval implements Serializable
{
   public Interval(Comparable first, Comparable second) { . . . }
   . . .
   private Comparable lower;
   private Comparable upper;
}
```

Возможно, вы спросите, что произойдет, если изменить порядок следования ограничивающих типов: `class Interval<Serializable & Comparable>`. В этом случае в низкоуровневом коде `T` заменится на `Serializable`, а компилятор будет при необходимости выполнять приведение к типу `Comparable`. Для повышения эффективности можно добавить в конец списка ограничивающих типов интерфейсы-маркеры (т.е. интерфейсы, в которых не объявлены методы).

Преобразование универсальных выражений

Когда в программе вызывается универсальный метод, компилятор выполняет преобразование в низкоуровневый код и включает операторы приведения возвращаемого типа. Рассмотрим, например, следующие выражения:

```
Pair<Employee> buddies = . . .;
Employee buddy = buddies.getFirst();
```

В результате преобразования в низкоуровневый код метод `getFirst()` возвращает тип `Object`. Компилятор автоматически преобразует возвращаемый тип в `Employee`. Таким образом, вызов метода представляется виртуальной машине двумя инструкциями.

- Вызов низкоуровневого метода `Pair.getFirst()`.
- Приведение возвращаемого значения `Object` к типу `Employee`.

Приведение типов также осуществляется при обращении к универсальным полям. Предположим, что поля `first` и `second` класса `Pair` объявлены как `public`. (Очевидно, что подобный стиль программирования не приветствуется, однако такой

код не противоречит правилам языка Java.) Если в программе имеется приведенное ниже выражение, то в результирующем байтовом коде также будет приведение типов.

```
Employee buddy = buddies.first;
```

Преобразование универсальных методов

Универсальные методы также преобразуются в низкоуровневый код. Обычно объявление универсального метода выглядит подобно выражению, приведенному ниже.

```
public static <T extends Comparable> T min(T[] a)
```

При этом разработчик считает, что имеет дело с целым семейством методов, однако после преобразования остается единственный метод.

```
public static Comparable min(Comparable[] a)
```

Обратите внимание на то, что параметр типа `T` удален; остался лишь ограничивающий тип `Comparable`.

Преобразование универсальных методов в низкоуровневый код связано с некоторыми сложностями. Рассмотрим следующий пример:

```
class DateInterval extends Pair<Date>
{
   public void setSecond(Date second)
   {
      if (second.compareTo(getFirst()) >= 0)
         super.setSecond(second);
   }
   . . .
}
```

Временной интервал представляет собой пару объектов `Date`, и нам надо переопределить методы, чтобы убедиться, что второе значение не меньше первого. После преобразования в низкоуровневый код класс выглядит следующим образом:

```
class DateInterval extends Pair // После преобразования
{
   public void setSecond(Date second) { . . . }
   . . .
}
```

Предположим, что существует еще один метод `setSecond()`, унаследованный от `Pair`.

```
public void setSecond(Object second)
```

Это совершенно другой метод, так как для него определен иной тип параметра — `Object` вместо `Date`. Однако он не должен быть другим. Рассмотрим приведенную ниже последовательность выражений.

```
DateInterval interval = new DateInterval(. . .);
Pair<Date> pair = interval; // Допустимо; присваивание
                            // переменной суперкласса
pair.setSecond(aDate);
```

Мы ожидаем, что в соответствии с принципом полиморфизма будет вызван требуемый метод. Поскольку переменная ссылается на объект `DateInterval`, это дол-

жен быть метод DateInterval.setSecond(). Однако удаление переменных типов конфликтует с принципами полиморфизма. Для того чтобы устранить проблему, компилятор генерирует в классе DateInterval специальный связующий метод (bridge method).

```
public void setSecond(Object second) { setSecond((Date) second); }
```

Чтобы понять, как данный подход позволяет решить проблему, проследим, как выполняется следующее выражение.

```
pair.setSecond(aDate)
```

Тип переменной pair объявлен как Pair<Date>, и для этого типа определен единственный метод setSecond, а именно setSecond(Object). Виртуальная машина вызывает метод объекта, на который ссылается pair. Данный объект имеет тип DateInterval. Следовательно, должен быть вызван метод DateInterval .setSecond(Object). Это сгенерированный связующий метод. Он вызывает DateInterval.setSecond(Date), т.е. происходит именно то, что требуется.

Связующие методы могут выполнять и более сложные действия. Предположим, что в классе DateInterval также переопределен метод getSecond().

```
class DateInterval extends Pair<Date>
{
   public Date getSecond() { return
      (Date) super.getSecond().clone(); }
   . . .
}
```

При преобразовании в низкоуровневый код мы получим два метода getSecond().

```
Date getSecond()     // Определен в DateInterval
Object getSecond()   // Определен в Pair
```

Такой код недопустим: в классе не должно быть двух методов с одинаковыми именами и набором параметров (в данном случае параметры отсутствуют). Однако в виртуальной машине метод определяется типами параметров и типом возвращаемого значения. Поэтому компилятор может создать байтовый код для двух методов, которые отличаются только типом возвращаемого значения, и виртуальная машина корректно будет вести себя в этой ситуации.

Область применения связующих методов не ограничивается универсальными типами. В главе 5 было сказано, что с версии JDK 5.0 при переопределении метода допустимо указывать более ограниченный класс возвращаемых значений. Рассмотрим следующий фрагмент кода:

```
public class Employee implements Cloneable
{
   public Employee clone() throws CloneNotSupportedException {
... }
}
```

Методы Object.clone() и Employee.clone() имеют ковариантные типы возвращаемых значений.

Реально класс `Employee` содержит два метода `clone()`.

```
Employee clone() // Определен выше
Object clone()   // Синтезированный связующий метод,
                 // переопределяет Object.clone()
```

Сгенерированный связующий метод вызывает вновь определенный метод.

В завершении разговора о преобразовании универсального кода Java сформулируем основные положения, которые следует запомнить.

- Виртуальная машина не поддерживает универсальный код. Она работает с обычными классами и методами.
- Все параметры типов заменяются ограничивающими типами.
- Связующие методы генерируются для того, чтобы не нарушались принципы полиморфизма.
- По мере необходимости автоматически выполняется приведение типов.

Обращение к существующему коду

Большое количество программ на Java было написано до появления JDK 5.0. Если бы универсальные классы не могли взаимодействовать с этим кодом, они не получили бы широкого распространения. К счастью, использовать универсальные классы совместно с низкоуровневыми вариантами из существующих API достаточно просто.

Рассмотрим конкретный пример. Для того чтобы установить метки объекта `JSlider`, мы используем приведенный ниже метод.

```
void setLabelTable(Dictionary table)
```

В главе 9 для заполнения таблицы меток применялся следующий код:

```
Dictionary<Integer, Component> labelTable =
   new Hashtable<Integer, Component>();
labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));
labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));
. . .
slider.setLabelTable(labelTable); // Предупреждение
```

Класс `Hashtable` — конкретный подкласс абстрактного класса `Dictionary`. И класс `Dictionary`, и класс `Hashtable` были объявлены устаревшими после того, как в JDK 1.2 были реализованы заменяющие их интерфейс `Map` и класс `HashMap`. Однако они пока еще поддерживаются, и при желании разработчик может использовать их в своей программе (хотя это и не рекомендуется). С другой стороны, класс `JSlider` был реализован лишь в JDK 1.3. Знали ли о нем программисты? Могли ли они предположить, что в скором времени появятся универсальные типы? Ответы на эти вопросы еще раз демонстрируют необходимость решения задач взаимодействия с существующим кодом.

В JDK 5.0 классы `Dictionary` и `Hashtable` были превращены в универсальные. Другими словами, мы можем вместо низкоуровневого класса `Dictionary` использовать `Dictionary<Integer, Component>`. Однако, если вы передадите методу

setLabelTable() объект Dictionary<Integer, Component>, компилятор отобразит предупреждающее сообщение.

```
Dictionary<Integer, Components> labelTable = . . .;
slider.setLabelTable(labelTable); // Предупреждение
```

Дело в том, что компилятор не знает, что метод setLabelTable() может сделать с объектом Dictionary. Например, он может заменить все ключи строками. Это нарушит соглашения о том, что все ключи должны иметь тип Integer, и при выполнении последующих операций возможны исключения вследствие некорректного приведения типов.

В ответ на это предупреждающее сообщение вы почти ничего не можете предпринять. Вам стоит разве что обдумать ситуацию и выяснить, что может сделать JSlider с объектом Dictionary. В нашем случае ясно, что JSlider лишь читает информацию, поэтому предупреждающее сообщение можно проигнорировать.

Теперь рассмотрим другой случай. Предположим, что вы получаете от уже существующего класса низкоуровневый объект. Вы можете присвоить его переменной, в которой задан параметр типа, но, очевидно, что снова получите предупреждение.

```
Dictionary<Integer, Components> labelTable =
   slider.getLabelTable(); // Предупреждение
```

Обдумайте сообщение и убедитесь в том, что таблица меток действительно содержит объекты Integer и Component. Очевидно, что абсолютную гарантию корректного поведения программы получить невозможно. Неопытный разработчик или злоумышленник может установить для линейного регулятора другой объект Dictionary. Однако ситуация не станет хуже той, которая была бы до появления JDK 5.0. В крайнем случае будет сгенерировано исключение.

К сожалению, изучив причины возникновения предупреждающих сообщений, отключить вывод этих сведений невозможно. А рассмотрение повторяющихся сообщений при каждой повторной компиляции исходного кода занимает много времени и не приносит никакой пользы. Разработчики языка Java планируют реализовать в последующих версиях JDK более гибкий механизм предупреждений.

Ограничения на работу с универсальными типами

В следующих разделах мы обсудим ряд ограничений, которые необходимо учитывать при работе с универсальными типами Java. Большинство из них является следствием преобразования в низкоуровневый код.

Простые типы

Вместо параметра типа нельзя подставить простой тип. Так, нельзя записать Pair<double>; допустимо лишь выражение Pair<Double>. Причина, конечно же, в преобразовании в низкоуровневый код. После преобразования класс Pair содержит поля типа Object, в которых нельзя хранить значения double и других простых типов.

Это, разумеется, неудобно, но вполне оправдано и согласуется со статусом простых типов в языке Java. Данный недостаток не так уж существен. Имеется всего восемь простых типов и, если интерфейсные классы вас по какой-то причине не устраивают, вы всегда можете использовать для обработки таких данных отдельные классы и методы.

Запрос типа в процессе выполнения программы

В виртуальной машине Java обрабатываются только обычные, не универсальные объекты. Поэтому в ответ на любой запрос вы получите низкоуровневый тип. Так, следующее выражение лишь проверяет, является ли a экземпляром класса `Pair`:

```
if (a instanceof Pair<String>) // То же что и a instanceof Pair
```

Также обстоит дело и с приведением типов.

```
Pair<String> p = (Pair<String>) a;
```

Чтобы напомнить разработчику о том, что подобное использование универсальных классов связано с риском, компилятор отображает предупреждающее сообщение.

Нетрудно догадаться, что метод `getClass()` также всегда возвращает низкоуровневый тип. Рассмотрим следующий фрагмент кода.

```
Pair<String> stringPair = . . .;
Pair<Employee> employeePair = . . .;
if (stringPair.getClass() == employeePair.getClass()) // Равенство
```

В данном примере в результате сравнения получается значение `true`, потому что оба вызова `getClass()` возвращают `Pair.class`.

Исключения

Сгенерировать или перехватить исключение, описываемое универсальным объектом, невозможно. По сути, недопустимо даже создавать универсальный класс, расширяющий `Throwable`. Например, следующее определение не будет скомпилировано:

```
public class Problem<T> extends Exception { /* . . . */ }
   // Ошибка: универсальный класс не может расширять Throwable
```

Нельзя использовать переменную типа в выражении `catch`. Так, например, при компиляции приведенного ниже метода вы получите сообщение об ошибке.

```
public static <T extends Throwable> void doWork(Class<T> t)
{
   try
   {
      do work
   }
   catch (T e) // Ошибка: переменная типа недопустима
               // в выражении catch
   {
      Logger.global.info(...)
   }
}
```

Однако в выражении `throws` переменная типа вполне допустима. Приведенный ниже код метода корректен.

```
public static <T extends Throwable> void doWork(T t) throws T
        // Допустимо
{
   try
   {
      do work
```

```
    }
    catch (Throwable realCause)
    {
       t.initCause(realCause);
       throw t;
    }
}
```

Массивы

Нельзя объявлять массивы с параметрами типов. Например, следующее выражение недопустимо:

```
Pair<String>[] table = new Pair<String>(10); // Ошибка
```

Почему? После преобразования в низкоуровневый код тип таблицы будет `Pair[]`. Его нельзя превратить в `Object[]`.

```
Object[] objarray = table;
```

В главе 5 было сказано, что массивы запоминают тип элементов и при попытке записать данные другого типа генерируют исключение `ArrayStoreException`.

```
objarray[0] = "Hello"; // Ошибка: тип элемента Pair
```

Однако преобразование универсальных типов в низкоуровневый код делает этот механизм неэффективным. Приведенное ниже выражение выдержит проверку, но присваиваемый тип все же будет некорректным.

```
objarray[0] = new Pair<Employee>();
```

По этой причине массивы, элементами которых являются универсальные типы, запрещены.

Если вам надо организовать хранение объектов с параметрами типов, используйте `ArrayList`: выражение `ArrayList<Pair<String>>` эффективно и безопасно.

Создание экземпляров универсальных типов

Создать экземпляр универсального типа невозможно. Так, например, приведенный ниже конструктор `Pair<T>` недопустим.

```
public Pair() { first = new T(); second = new T(); } // Ошибка
```

В результате преобразования в низкоуровневый код `T` заменится на `Object`, а выражение `new Object()` некорректно.

Нельзя также создать универсальный массив.

```
public <T> T[] minMax(T[] a) { T[] mm = new T[2]; . . . } // Ошибка
```

На низком уровне будет предпринята попытка сформировать массив `Object[2]`.

Однако универсальные объекты и массивы можно создавать средствами отражения. Для этого надо использовать методы `Class.newInstance()` и `Array.newInstance()`.

Статический контекст

Нельзя ссылаться на переменные типа в статических полях и методах. Например, следующий фрагмент кода не будет работать, хотя на первый взгляд подобное решение может показаться привлекательным:

```
public class Singleton<T>
{
   public static T getSingleInstance() // Ошибка
   {
      if (singleInstance == null) // Создание нового экземпляра T
      return singleInstance;
   }
   private static T singleInstance; // Ошибка
}
```

Если бы он мог быть выполнен, в программе можно было бы объявить `Singleton<Random>` для совместного использования генератора случайных чисел и `Singleton<JFileChooser>` для разделения диалогового окна выбора файлов. Однако работать он не будет. После преобразования в низкоуровневый код останется лишь один класс `Singleton` и одно поле `singleInstance`. По этой причине статические поля и методы с переменными типа запрещены.

Конфликты в результате приведения к низкоуровневому коду

Недопустимо, чтобы после преобразования универсальных типов в низкоуровневый код возникали условия, способные вызывать конфликты. Рассмотрим пример. Предположим, что в класс `Pair` добавлен метод `equals()`.

```
public class Pair<T>
{
   public boolean equals(T value)
      { return first.equals(value) && second.equals(value); }
   . . .
}
```

Рассмотрим выражение `Pair<String>`. Очевидно, что полученный класс содержит два метода `equals()`.

```
boolean equals(String) // Определен в Pair<T>
boolean equals(Object) // Унаследован от Object
```

Однако интуитивно ясно, что этого не должно быть. В результате преобразования в низкоуровневый код метод

```
boolean equals(T)
```

превратится в

```
boolean equals(Object)
```

который будет конфликтовать с методом `Object.equals()`.

Выход — переименовать конфликтующий метод.

В описании универсальных типов сформулировано следующее правило. "Для поддержки преобразования в низкоуровневый код налагается следующее ограничение: класс или переменная типа не может одновременно быть подтипом двух интерфей-

сов, которые сводятся к одному интерфейсу." Например, следующие выражения некорректны:

```
class Calendar implements Comparable<Calendar> { . . . }
class GregorianCalendar extends Calendar
   implements Comparable<GregorianCalendar> { . . . } // Ошибка
```

Здесь `GregorianCalendar` реализует как `Comparable<Calendar>`, так и `Comparable<GregorianCalendar>`, которые, по сути, являются различными представлениями одного и того же интерфейса.

Не совсем ясно, как должно повлиять данное ограничение на преобразование кода. Обычные варианты тех же классов вполне допустимы.

```
class Employee implements Comparable { . . . }
class Manager extends Employee implements Comparable { . . . }
```

Правила наследования для универсальных типов

При работе с универсальными классами необходимо выполнять правила, касающиеся наследования и создания подтипов. Рассмотрим ситуацию, которую многие программисты попросту не учитывают. Пусть в нашем распоряжении есть класс и его подкласс, например `Employee` и `Manager`. Является ли `Pair<Manager>` подклассом `Pair<Employee>`? Может показаться странным, но ответ на этот вопрос отрицательный. Например, приведенный ниже код не будет скомпилирован.

```
Manager[] topHonchos = . . .;
Pair<Employee> = ArrayAlg.minmax(topHonchos); // Ошибка
```

Метод `minmax()` возвращает `Pair<Manager>`, а не `Pair<Employee>`, и недопустимо присваивать переменной одного типа значение другого типа.

Между `Pair<S>` и `Pair<T>` нет никакой связи, независимо от того, связаны ли друг с другом типы `S` и `T` (рис. 13.1).

Такое ограничение выглядит неоправданно жестким, но оно необходимо в целях безопасности. Предположим, что возможно преобразовать `Pair<Manager>` в `Pair<Employee>`, и рассмотрим следующий фрагмент кода:

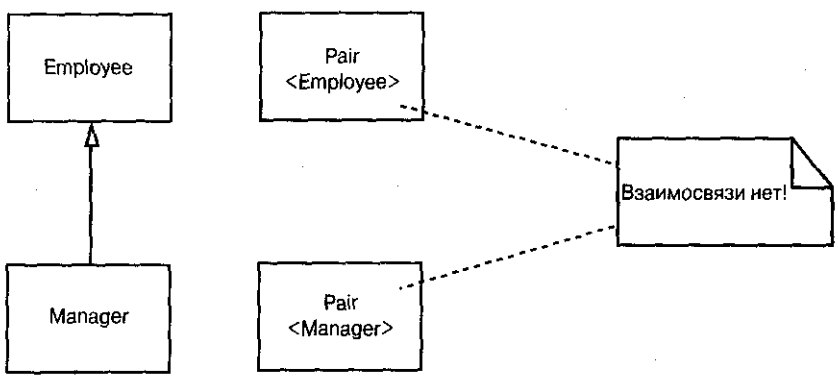


Рис. 13.1. Класс `Pair<Manager>` не является подклассом `Pair<Employee>`

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair<Employee> employeeBuddies = managerBuddies; // Недопустимо,
                    // но предположим, что так можно поступить
employeeBuddies.setFirst(lowlyEmployee);
```

Очевидно, что последнее выражение корректно. Однако employeeBuddies и managerBuddies ссылаются на один и тот же объект. Другими словами, мы собираемся объединить руководителя с обычным сотрудником, что невозможно для Pair<Manager>.

В случае необходимости вы всегда сможете преобразовать параметризованный тип в низкоуровневый. Например, Pair<Employee> — это подтип низкоуровневого типа Pair. *Такое преобразование необходимо для взаимодействия с существующим кодом.*

Можно ли выполнить преобразование в низкоуровневый тип так, что возникнет ошибка? К сожалению, это возможно. Рассмотрим следующий пример:

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair rawBuddies = managerBuddies;        // Допустимо
rawBuddies.setFirst(new File(". . ."));  // Только предупреждение
                                         // на этапе компиляции
```

Такая перспектива выглядит устрашающе. Однако не забывайте, что те же ошибки могут возникнуть при использовании предыдущих версий Java. Защита виртуальной машины не улучшилась. Когда посторонний объект извлекается с помощью метода getFirst() и присваивается переменной, генерируется, как и положено, исключение ClassCastException. Вы лишь теряете дополнительный уровень защиты, который в нормальных условиях обеспечивают универсальные типы.

Универсальные классы могут расширять или реализовывать другие универсальные классы. В этом смысле они не отличаются от привычных всем классов. Например, класс ArrayList<T> реализует интерфейс List<T>. Это означает, что ArrayList<Manager> может быть преобразован в List<Manager>. Однако, как вы уже знаете, ArrayList<Manager> — это не ArrayList<Employee> и не List<Employee>. Взаимосвязь между указанными классами и интерфейсами показана на рис. 13.2.

Подстановочные типы

По мнению исследователей, универсальным типам в том виде, в котором мы их до сих пор рассматривали, недостает гибкости. Разработчики языка Java придумали очень остроумный и в то же время безопасный механизм *подстановочных типов.* Например, приведенное ниже выражение означает любой универсальный тип Pair, параметр которого представляет собой подкласс Employee.

```
Pair<? extends Employee>
```

Так, в данном случае допустимо EmployeePair<Manager>, но не Pair<String>.

Предположим, что вы хотите создать метод, который выводил бы информацию о парах сотрудников. Этот метод может иметь следующий вид:

```
public static void printBuddies(Pair<Employee> p)
{
   Employee first = p.getFirst();
   Employee second = p.getSecond();
   System.out.println(first.getName() + " and " +
      second.getName() + " are buddies.";
}
```

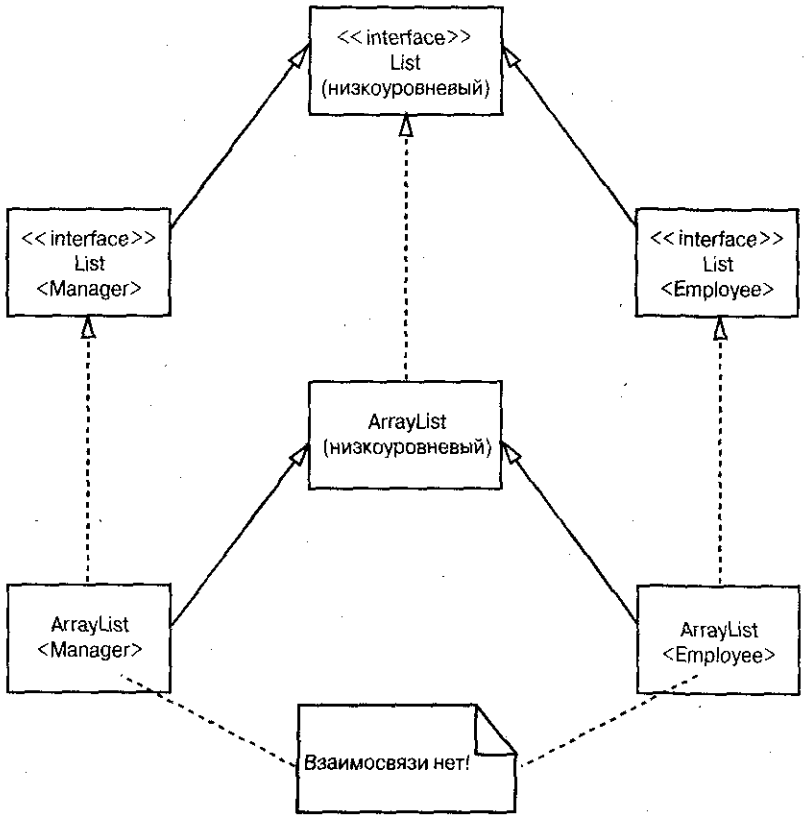


Рис. 13.2. Взаимодействие универсальных типов

Как было сказано в предыдущем разделе, этому методу нельзя передать `Pair<Manager>`, что ограничивает его применимость. Решение проблемы состоит в использовании подстановочных типов.

```
public static void printBuddies(Pair<? extends Employee> p)
```

Тип `Pair<Manager>` является подтипом `Pair<? extends Employee>` (рис. 13.3).

Можно ли использовать подстановочные типы, чтобы разрушить `Pair<Manager>` посредством ссылки `Pair<? extends Employee>`?

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair<? extends Employee> wildcardBuddies = managerBuddies; // Допустимо
wildcardBuddies.setFirst(lowlyEmployee); // Ошибка на этапе компиляции
```

Это невозможно. Обращение к `setFirst()` повлечет ошибку, связанную с неправильным использованием типов. Чтобы понять, почему это произойдет, рассмотрим более подробно тип `Pair<? extends Employee>`. Методы этого класса выглядят следующим образом:

```
? extends Employee getFirst()
void setFirst(? extends Employee)
```

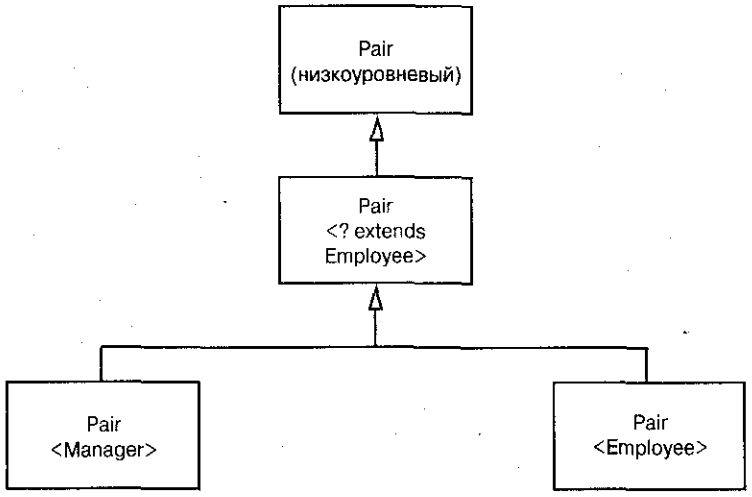


Рис. 13.3. Взаимосвязь между подтипами, определяемыми посредством подстановочных типов

В результате обращение к методу `setFirst()` становится невозможным. Компилятор лишь знает, что необходимо использовать некоторый подтип `Employee`, но не знает, какой это должен быть тип. Он отказывается передавать конкретный тип, так как ? может не соответствовать ему.

С методом `getFirst()` подобная проблема не возникает. Вполне допустимо присвоить значение, возвращаемое этим методом, переменной `Employee`.

На этом основывается использование ограниченных подстановочных типов. Теперь у нас есть способ разделять методы доступа, не представляющие опасности для программы, и модифицирующие методы.

Ограничения супертипов

Ограниченные подстановочные типы напоминают ограниченные универсальные типы, но для них реализована дополнительная возможность: вы можете задавать ограничения супертипов. Это осуществляется посредством следующего выражения:

```
? super Manager
```

Этот подстановочный тип ограничен всеми супертипами `Manager`. (Тот факт, что существующее ключевое слово `super` точно описывает данную взаимосвязь, можно рассматривать как неожиданную удачу.)

Зачем это надо? Подстановочные типы с ограничением на супертипы позволяют реализовать поведение программы, противоположное рассмотренному ранее. Вы можете передать параметры методам, но не использовать возвращаемые значения. Например, `Pair<? super Manager>` содержит следующие методы:

```
void set(? super Manager)
? super Manager get()
```

Компилятор не знает точный тип для метода `set()`, но может передавать ему любые объекты `Manager` (или их подтипы, например, `Executive`), но не объект `Employee`. Однако при вызове метода `get()` нет никаких гарантий относительно типа возвращаемого объекта. Его можно присвоить только переменной `Object`.

Ниже приведен типичный пример, иллюстрирующий сказанное. В нашем распоряжении есть массив объектов, описывающих менеджеров; объекты с минимальной и максимальной премией помещаются в объект `Pair`. Какой вариант `Pair` будет использован? Подходит `Pair<Employee>` или `Pair<Object>` (рис. 13.4). Следующий метод принимает любой подходящий вариант `Pair`:

```
public static void minMaxBonus(Manager[] a,
   Pair<? super Manager> result)
{
   if (a == null || a.length == 0) return;
   Manager min = a[0];
   Manager max = a[0];
   for (int i = 1; i < a.length; i++)
   {
      if (min.getBonus() > a[i].getBonus()) min = a[i];
      if (max.getBonus() < a[i].getBonus()) max = a[i];
   }
   result.setFirst(min);
   result.setSecond(max);
}
```

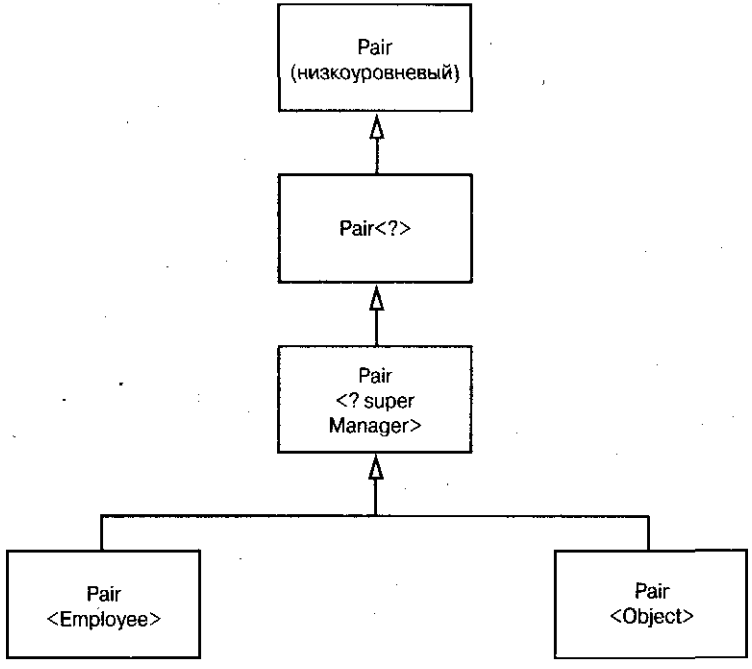


Рис. 13.4. Подстановочный тип с ограничением супертипа

Как следует из вышесказанного, подстановочные типы с ограничениями на суперклассы позволяют записывать в универсальные объекты, а подстановочные типы с ограничениями на подклассы — читать из универсальных объектов.

Однако ограничения на суперклассы могут быть использованы и по-другому. Рассмотрим интерфейс `Comparable`. Он сам является универсальным типом и объявлен следующим образом:

```
public interface Comparable<T>
{
    public int compareTo(T other);
}
```

Здесь переменная типа определяет тип параметра `other`. Например, класс `String` реализует интерфейс `Comparable<String>`, соответственно его метод `compareTo()` выглядит следующим образом:

```
public int compareTo(String other)
```

В данном случае явный параметр имеет правильный тип. До появления JDK 5.0 параметр `other` имел тип `Object` и в реализации метода необходимо было приведение типов.

Мы можем использовать тот факт, что `Comparable` является универсальным типом при написании метода `min()` класса `ArrayAlg`. Этот метод можно объявить так, как показано ниже.

```
public static <T extends Comparable<T>> T min(T[] a) . . .
```

Такой подход выглядит лучше, чем обычное использование выражения `T extends Comparable`; и действительно, для многих классов он дает хорошие результаты. Например, если вычисляется минимум для массива `String`, `T` превращается в тип `String`, а `String` является подтипом `Comparable<String>`. Однако при обработке массива объектов `GregorianCalendar` возникает проблема. `GregorianCalendar` представляет собой подкласс `Calendar`, а `Calendar` реализует `Comparable<Calendar>`. Таким образом, `GregorianCalendar` реализует `Comparable<Calendar>`, но не `Comparable<GregorianCalendar>`.

В подобной ситуации приходят на помощь супертипы.

```
public static <T extends Comparable<? super T>> T min(T[] a) . . .
```

Теперь метод `compareTo()` приобретает следующий вид:

```
int compareTo(? super T)
```

Он получает объект типа `T`, или, как в случае, когда `T` — это `GregorianCalendar`, супертипа `T`. В любом случае допустимо передавать объект `T` методу `compareTo()`.

До инициализации объявление типа `<T extends Comparable<? super T>>` выглядит устрашающе. Это, конечно, недостаток, поскольку данное объявление призвано помочь разработчикам приложений, удаляя необязательные ограничения параметров. Программисты, специализирующиеся на создании прикладных программ, быстро научатся использовать подобные объявления, предполагая, что разработчик библиотеки все сделал правильно. Если же вы создаете библиотеки, вам следует досконально разобраться в особенностях подстановочных типов, в противном случае вы получите дурную репутацию у пользователей, которые будут пытаться выполнить приведение типов, чтобы компилятор обработал созданный ими код.

Подстановочные типы без ограничений

При необходимости вы можете использовать подстановочные типы, не указывая никаких ограничений (например, `Pair<?>`). На первый взгляд такая конструкция будет действовать как низкоуровневый тип `Pair`. Однако на самом деле ситуация несколько иная. Тип `Pair<?>` имеет следующие методы:

```
? getFirst()
void setFirst(?)
```

Возвращаемое значение метода `getFirst()` может быть присвоено только переменной типа `Object`. Метод `setFirst()` не будет вызван никогда, даже при передаче ему параметра `Object`. В этом существенное отличие между `Pair<?>` и `Pair`: метод `setObject()` низкоуровневого класса `Pair` можно вызвать с любой ссылкой на объект `Object`.

Зачем может понадобиться подобный специфический тип? Он подходит для очень простых операций. Предположим, что метод, код которого приведен ниже, проверяет, содержится ли в паре требуемый объект. Реальный тип в данном случае не нужен.

```
public static boolean hasNulls(Pair<?> p)
{
   return p.getFirst() == null || p.getSecond() == null;
}
```

Избежать применения подстановочного типа можно, превратив метод в универсальный.

```
public static <T> boolean hasNulls(Pair<T> p)
```

Однако вариант с подстановочным типом проще для восприятия.

Перехват подстановочного типа

Создадим метод, который переставлял бы местами элементы пары.

```
public static void swap(Pair<?> p)
```

Знак подстановочного типа не является переменной типа, поэтому мы не можем написать код, в котором присутствовал бы тип `?`. Другими словами, следующий фрагмент кода недопустим:

```
? t = p.getFirst(); // Ошибка
p.setFirst(p.getSecond());
p.setSecond(t);
```

Это существенное затруднение, поскольку в процессе перестановки нам надо временно хранить первый элемент. К счастью, для этой проблемы есть интересное решение. Мы можем написать вспомогательный метод `swapHelper()`, подобный представленному ниже.

```
public static <T> void swapHelper(Pair<T> p)
{
   T t = p.getFirst();
   p.setFirst(p.getSecond());
   p.setSecond(t);
}
```

Заметьте, что `swapHelper()` — универсальный метод, а `swap()` — нет; он имеет фиксированный параметр типа `Pair<?>`.

Теперь из метода `swap()` можно вызвать `swapHelper()`.

```
public static void swap(Pair<?> p) { swapHelper(p); }
```

В данном случае параметр `T` метода `swapHelper()` перехватывает подстановочный тип. Неизвестно, какой тип обозначается с помощью подстановочного, однако этот некоторый конкретный тип и определение `<T>swapHelper` имеет смысл в случае, когда `T` обозначает этот тип.

В данном случае нельзя утверждать, что мы вынуждены использовать подстановочный тип — не произойдет ничего плохого, если мы применим параметр типа, как в методе `swapHelper()`. Однако рассмотрим другой пример, в котором подстановочный тип естественным образом возникает в процессе вычислений.

```
public static void maxMinBonus(Manager[] a,
   Pair<? super Manager> result)
{
   minMaxBonus(a, result);
   PairAlg.swapHelper(result); // swapHelper перехватывает
                               // подстановочный тип
}
```

Здесь нельзя обойтись без перехвата подстановочного типа.

Перехват подстановочного типа имеет лишь ограниченное применение. Компилятор должен гарантировать, что подстановочный тип представляет единственный определенный тип. Например, `T` в `ArrayList<Pair<T>>` не может перехватывать подстановочный тип в `ArrayList<Pair<?>>`. Списочный массив может содержать два объекта `Pair<?>`, в которых знак `?` представляет различные типы.

Программа, код которой представлен в листинге 13.3, демонстрирует различные подходы, описанные ранее.

Листинг 13.3. Содержимое файла `PairTest3.java`

```
import java.util.*;

public class PairTest3
{
   public static void main(String[] args)
   {
      Manager ceo = new Manager("Gus Greedy", 800000,
         2003, 12, 15);
      Manager cfo = new Manager("Sid Sneaky", 600000,
         2003, 12, 15);
      Pair<Manager> buddies = new Pair<Manager>(ceo, cfo);
      printBuddies(buddies);

      ceo.setBonus(1000000);
      cfo.setBonus(500000);
      Manager[] managers = { ceo, cfo };

      Pair<Employee> result = new Pair<Employee>();
      minMaxBonus(managers, result);
      System.out.println("first: " + result.getFirst().getName()
```

```
            + ", second: " + result.getSecond().getName());
         maxMinBonus(managers, result);
         System.out.println("first: " + result.getFirst().getName()
            + ", second: " + result.getSecond().getName());
      }

      public static void printBuddies(Pair<? extends Employee> p)
      {
         Employee first = p.getFirst();
         Employee second = p.getSecond();
         System.out.println(first.getName() + " and " +
            second.getName() + " are buddies.");
      }

      public static void minMaxBonus(Manager[] a,
         Pair<? super Manager> result)
      {
         if (a == null || a.length == 0) return;
         Manager min = a[0];
         Manager max = a[0];
         for (int i = 1; i < a.length; i++)
         {
            if (min.getBonus() > a[i].getBonus()) min = a[i];
            if (max.getBonus() < a[i].getBonus()) max = a[i];
         }
         result.setFirst(min);
         result.setSecond(max);
      }

      public static void maxMinBonus(Manager[] a,
         Pair<? super Manager> result)
      {
         minMaxBonus(a, result);
         PairAlg.swapHelper(result); // swapHelper перехватывает
                                     // подстановочный тип.
      }
   }

   class PairAlg
   {
      public static boolean hasNulls(Pair<?> p)
      {
         return p.getFirst() == null || p.getSecond() == null;
      }

      public static void swap(Pair<?> p) { swapHelper(p); }

      public static <T> void swapHelper(Pair<T> p)
      {
         T t = p.getFirst();
         p.setFirst(p.getSecond());
         p.setSecond(t);
      }
   }

   class Employee
```

```
{
   public Employee(String n, double s,
      int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar =
         new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   private String name;
   private double salary;
   private Date hireDay;
}

class Manager extends Employee
{
   /**
      @param n Имя сотрудника
      @param s Зарплата
      @param year Год найма
      @param month Месяц найма
      @param day День найма
   */
   public Manager(String n, double s,
      int year, int month, int day)
   {
      super(n, s, year, month, day);
      bonus = 0;
   }

   public double getSalary()
   {
      double baseSalary = super.getSalary();
```

```
        return baseSalary + bonus;
    }

    public void setBonus(double b)
    {
        bonus = b;
    }

    public double getBonus()
    {
        return bonus;
    }

    private double bonus;
}
```

Отражение и универсальные типы

В настоящее время класс `Class` является универсальным. Например, `String.class` – это экземпляр (причем единственный) класса `Class<String>`. Приведенные ниже методы `Class<T>` используют параметры типов.

```
T newInstance()
T cast(Object obj)
T[] getEnumConstants()
Class<? super T> getSuperclass()
Constructor<T> getConstructor(Class... parameterTypes)
Constructor<T> getDeclaredConstructor(Class... parameterTypes)
```

Метод `newInstance()` возвращает экземпляр класса, полученный от конструктора по умолчанию. Его возвращаемый тип должен быть объявлен как `T`, т.е. быть тем же классом, который описывается с помощью `Class<T>`. Это предотвращает приведение типов.

Метод `cast()` возвращает заданный объект как тип `T`. Это происходит при условии, что тип объекта является подтипом `T`. В противном случае генерируется исключение `BadCastException`.

Метод `getEnumConstants()` возвращает значение `null`, если данный класс не является классом `enum` или массивом нумерованных значений, о которых известно, что они принадлежат типу `T`.

И наконец, методы `getConstructor()` и `getDeclaredConstructor()` возвращают объект `Constructor<T>`. Класс `Constructor` также стал универсальным, поэтому его метод `newInstance()` возвращает значение корректного типа.

`java.lang.Class<T>` 1.0

- `T newInstance()` 5.0

  Возвращает новый объект, созданный с помощью конструктора по умолчанию.

- `T cast(Object obj)` 5.0

  Возвращает `obj` в случае значения `null` или если объект может быть приведен к типу `T`. В противном случае генерируется исключение `BadCastException`.

- `T[] getEnumConstants()` 5.0

Возвращает массив всех значений, если T — нумерованный тип, и значение null в противном случае.

- `Class<? super T> getSuperclass()` 5.0

  Возвращает суперкласс данного класса или значение null, если T — не класс или класс Object.

- `Constructor<T> getConstructor(Class... parameterTypes)` 5.0.
- `Constructor<T> getDeclaredConstructor(Class... parameterTypes)` 5.0

  Предоставляет общедоступный конструктор либо конструктор с указанными типами параметров.

java.lang.reflect.Constructor<T> 1.1

- `T newInstance(Object... parameters)` 5.0

  Возвращает новый объект, созданный с указанными параметрами.

Использование параметров Class<T> для проверки типов

В некоторых случаях бывает полезно проверить переменную типа Class<T> в универсальном методе. Рассмотрим следующий пример:

```
public static <T> Pair<T> makePair(Class<T> c)
   throws InstantiationException, IllegalAccessException
{
   return new Pair<T>(c.newInstance(), c.newInstance());
}
```

При вызове приведенного ниже метода Employee.class представляет собой экземпляр Class<Employee>.

```
makePair(Employee.class)
```

Параметр типа T в методе makePair() соответствует Employee, и компилятор может сделать вывод, что метод возвращает значение Pair<Employee>.

Информация об универсальных типах в виртуальной машине Java

Одна из уже известных вам особенностей универсальных классов состоит в том, что при передаче виртуальной машине Java происходит преобразование в низкоуровневый код. Возможно, вы удивитесь, узнав о том, что, несмотря на удаление типов, некоторая информация о них остается в памяти. Например, низкоуровневый класс Pair знает, что он был сгенерирован на базе универсального класса Pair<T>, но не имеет сведений о том, был ли это Pair<String> или Pair<Employee>.

Рассмотрим следующий метод:

```
public static Comparable min(Comparable[] a)
```

Он получается в результате преобразования в низкоуровневый код метода, описанного ниже.

```
public static <T extends Comparable<? super T>> T min(T[] a)
```

Используя расширенный API отражения, реализованный в JDK 5.0, вы можете выяснить следующее.

- Универсальный метод содержит параметр типа T.
- Параметр типа ограничен классом, который в свою очередь является универсальным.
- Ограничивающий тип содержит параметр подстановочного типа.
- Для подстановочного типа использовано ограничение суперкласса.
- Универсальный метод получает в качестве параметра универсальный массив.

Другими словами, вы можете восстановить некоторые сведения об универсальных классах и методах, объявленных разработчиком. Однако вы не в состоянии выяснить способ замены параметра типа для объектов или вызовов методов.

Информация о типе содержится в файлах классов и специально предназначена для того, чтобы разрешить отражение универсальных классов, несовместимых с ранними версиями виртуальных машин.

Для представления определений универсальных типов в JDK 5.0 был предложен новый интерфейс `Type`, принадлежащий пакету `java.lang.reflect`. Существуют следующие подтипы данного интерфейса.

- Класс `Class`, описывающий конкретные типы.
- Интерфейс `TypeVariable`, описывающий переменные типа (например, `T extends Comparable<? super T>`).
- Интерфейс `WildcardType`, описывающий подстановочные типы (например, `? super T`).
- Интерфейс `ParameterizedType`, описывающий универсальный класс или интерфейсы (например, `Comparable<? super T>`).
- Интерфейс `GenericArrayType`, описывающий универсальные массивы (например, `T[]`).

На рис. 13.5 представлена иерархия наследования для указанных типов. Заметьте, что четыре подтипа являются интерфейсами — виртуальная машина создает экземпляры классов, реализующих эти интерфейсы.

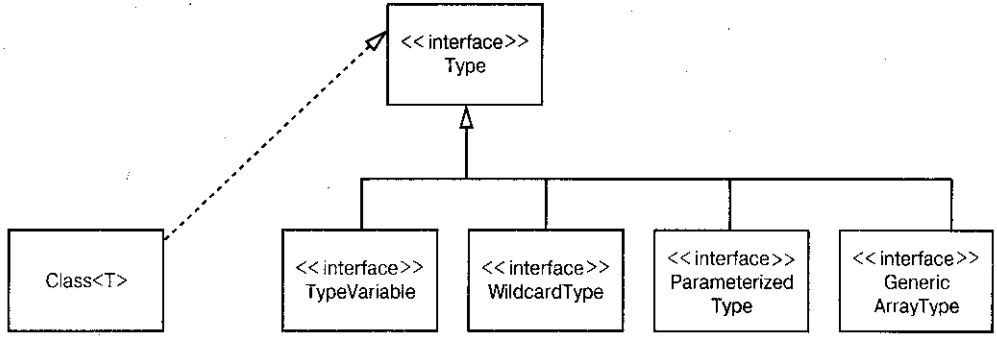


Рис. 13.5. Интерфейс Type и его дочерние интерфейсы

В листинге 13.4 показан код программы, которая выводит с помощью API отражения информацию о классе. Если вы зададите класс `Pair`, то получите следующие сведения:

```
class Pair<T extends java.lang.Object> extends java.lang.Object
public T extends java.lang.Object getSecond()
public void setFirst(T extends java.lang.Object)
public void setSecond(T extends java.lang.Object)
public T extends java.lang.Object getFirst()
```

Если вы зададите класс `ArrayAlg` в каталоге `PairTest2`, то получите приведенную ниже информацию о методе.

```
public static <T extends java.lang.Comparable> Pair<T extends
java.lang.Comparable> minmax(T extends java.lang.Comparable[])
```

Методы, используемые в программе, подробно описаны в документации по API.

Листинг 13.4. Содержимое файла `GenericReflectionTest.java`

```
import java.lang.reflect.*;
import java.util.*;

public class GenericReflectionTest
{
   public static void main(String[] args)
   {
      // Получение имени класса, заданного в командной строке.
      String name;
      if (args.length > 0)
         name = args[0];
      else
      {
         Scanner in = new Scanner(System.in);
         System.out.println
            ("Enter class name (e.g. java.util.Date): ");
         name = in.next();
      }

      try
      {
         // Вывод информации о классе и общедоступных методах
         Class cl = Class.forName(name);
         printClass(cl);
         for (Method m : cl.getDeclaredMethods())
            printMethod(m);
      }
      catch (ClassNotFoundException e)
      {
         e.printStackTrace();
      }
   }

   public static void printClass(Class cl)
   {
      System.out.print(cl);
```

```
      printTypes(cl.getTypeParameters(), "<", ", ", ">");
      Type sc = cl.getGenericSuperclass();
      if (sc != null)
      {
         System.out.print(" extends ");
         printType(sc);
      }
      printTypes(cl.getGenericInterfaces(),
         " implements ", ", ", "");
      System.out.println();
   }

   public static void printMethod(Method m)
   {
      String name = m.getName();
      System.out.print(Modifier.toString(m.getModifiers()));
      System.out.print(" ");
      printTypes(m.getTypeParameters(), "<", ", ", "> ");

      printType(m.getGenericReturnType());
      System.out.print(" ");
      System.out.print(name);
      System.out.print("(");
      printTypes(m.getGenericParameterTypes(), "", ", ", "");
      System.out.println(")");
   }

   public static void printTypes(Type[] types,
      String pre, String sep, String suf)
   {
      if (types.length > 0) System.out.print(pre);
      for (int i = 0; i < types.length; i++)
      {
         if (i > 0) System.out.print(sep);
         printType(types[i]);
      }
      if (types.length > 0) System.out.print(suf);
   }

   public static void printType(Type type)
   {
      if (type instanceof Class)
      {
         Class t = (Class) type;
         System.out.print(t.getName());
      }
      else if (type instanceof TypeVariable)
      {
         TypeVariable t = (TypeVariable) type;
         System.out.print(t.getName());
         printTypes(t.getBounds(), " extends ", " & ", "");
      }
      else if (type instanceof WildcardType)
      {
         WildcardType t = (WildcardType) type;
         System.out.print("?");
```

```
            printTypes(t.getLowerBounds(), " extends ", " & ", "");
            printTypes(t.getUpperBounds(), " super ", " & ", "");
         }
         else if (type instanceof ParameterizedType)
         {
            ParameterizedType t = (ParameterizedType) type;
            Type owner = t.getOwnerType();
            if (owner != null)
               { printType(owner); System.out.print("."); }
            printType(t.getRawType());
            printTypes(t.getActualTypeArguments(), "<", ", ", ">");
         }
         else if (type instanceof GenericArrayType)
         {
            GenericArrayType t = (GenericArrayType) type;
            System.out.print("");
            printType(t.getGenericComponentType());
            System.out.print("[]");
         }

      }
}
```

API **java.lang.Class<T> 1.0**

- `TypeVariable[] getTypeParameters()` **5.0**

  Если тип объявлен как универсальный, данный метод предоставляет переменные этого типа. В противном случае возвращается массив нулевой длины.

- `Type getGenericSuperclass()` **5.0**

  Предоставляет универсальный тип суперкласса, объявленного для данного типа. Если тип соответствует `Object` или не является классом, возвращается значение `null`.

- `Type[] getGenericInterfaces()` **5.0**

  Предоставляет универсальные типы интерфейсов, объявленных для данного типа. Порядок следования интерфейсов соответствует порядку их объявления. Если данный тип не реализует ни одного интерфейса, возвращается массив нулевой длины.

API **java.lang.reflect.Method 1.1.**

- `TypeVariable[] getTypeParameters()` **5.0**

  Предоставляет переменные универсального типа, если метод объявлен как универсальный. В противном случае возвращается массив нулевой длины.

- `Type getGenericReturnType()` **5.0**

  Предоставляет универсальный тип возвращаемого значения, объявленный для данного метода.

- `Type[] getGenericParameterTypes()` **5.0**

Предоставляет универсальные типы параметров, объявленные для данного метода. Если метод не имеет параметров, возвращается массив нулевой длины.

java.lang.reflect.TypeVariable 5.0

- `String getName()`

  Предоставляет имя переменной типа.

- `Type[] getBounds()`

  Предоставляет ограничения подкласса для данной переменной типа. Если переменная не ограничена, возвращается массив нулевой длины.

java.lang.reflect.WildcardType 5.0

- `Type[] getLowerBounds()`

  Предоставляет ограничения подкласса для данной переменной типа. Если ограничения подкласса отсутствуют, то возвращается массив нулевой длины.

- `Type[] getUpperBounds()`

  Предоставляет ограничения суперкласса для данной переменной типа. Если ограничения суперкласса отсутствуют, то возвращается массив нулевой длины.

java.lang.reflect.ParameterizedType 5.0

- *`Type getRawType()`*

  Предоставляет низкоуровневый тип для данного параметризованного типа.

- `Type[] getActualTypeArguments()`

  Предоставляет типы параметров, которые были объявлены.

- `Type getOwnerType()`

  Для внутреннего типа предоставляет внешний класс, для типа верхнего уровня возвращает значение `null`.

java.lang.reflect.GenericArrayType 5.0

- `Type getGenericComponentType()`

  Предоставляет тип универсального компонента, объявленного для массива.

Вы прочитали до конца первый том книги. В этом томе были описаны основы языка программирования Java и элементы стандартной библиотеки, необходимые при работе над подавляющим большинством программных проектов. Мы надеемся, *что вы легко восприняли изложенный материал и готовы применить полученные* знания на практике. Более сложные вопросы, такие как организация сетевого взаимодействия, поддержка нескольких потоков, обеспечение безопасности и интернационализация, будут рассмотрены во втором томе.

Приложение А

Ключевые слова

Ключевое слово	Описание	Глава
`absract`	Абстрактный класс или метод	5
`assert`	Используется для локализации ошибок в программе	11
`boolean`	Логический тип	3
`break`	Выход из оператора `switch` или цикла	3
`byte`	8-битовое целое число	3
`case`	Метка оператора `switch`	3
`catch`	Блок, следующий за блоком `try` и предназначенный для обработки исключения	11
`char`	Символьный тип в кодировке Unicode	3
`class`	Определяет тип класса	4
`const`	Не используется	
`continue`	Оператор продолжения в цикле	3
`default`	Раздел по умолчанию в операторе `switch`	3
`do`	Начало цикла `do/while`	3
`double`	Число с плавающей точкой двойной точности	3
`else`	Раздел `else` в операторе `if`	3
`extends`	Определяет родительский класс данного класса	4
`final`	Константа, класс, для которого запрещено создание подклассов, или метод, который нельзя переопределять	5
`finally`	Блок, используемый совместно с блоком `try`. Содержащиеся в нем выражения выполняются всегда	11
`float`	Число с плавающей точкой одинарной точности	3
`for`	Разновидность цикла	3
`goto`	Не используется	
`if`	Условный оператор	3
`implements`	Определяет интерфейс, реализуемый классом	6
`import`	Импортирует пакет	4
`instanceof`	Проверяет, является ли объект экземпляром класса	5

Ключевое слово	Описание	Глава
`interface`	Полностью абстрактный класс, в котором объявлены методы, определяемые в классе, реализующем данный интерфейс	6
`long`	64-битовое целое число	3
`native`	Платформенно-ориентированный метод (см. том 2)	
`new`	Создает новый объект или массив	3
`null`	Нулевая ссылка	3
`package`	Пакет классов	4
`private`	Элемент, доступный только методам данного класса	4
`protected`	Элемент, доступный только методам данного класса, его дочерним классам и другим классам в том же пакете	5
`public`	Элемент, доступный методам всех классов	4
`return`	Возвращает результат работы метода	3
`short`	16-битовое целое число	3
`static`	Элемент, принадлежащий классу, но не его экземплярам	3
`strictfp`	Использует строгие правила при вычислениях чисел с плавающей точкой	2
`super`	Объект или конструктор суперкласса	5
`switch`	Оператор выбора	3
`synchronized`	Метод или блок кода, являющийся атомарным для потока	(см. том 2)
`this`	Неявный аргумент метода или конструктора данного класса	4
`throw`	Генерирует исключение	11
`throws`	Объявляет исключения, которые могут генерироваться данным методом	11
`transient`	Отмечает данные, которые не должны быть постоянными	12
`try`	Блок, участвующий в перехвате исключений	11
`void`	Обозначает метод, который не возвращает никакого значения	3
`volatile`	Обеспечивает нескольким потокам согласованный доступ к полю	(см. том 2)
`while`	Разновидность цикла	3

Приложение B

Особенности кода, ориентированного на JDK 5.0

Программы, приведенные в этой книге в качестве примеров, в основном ориентированы на JDK 5.0. В данном приложении описаны изменения, которые надо внести в код программ, чтобы они компилировались и выполнялись с использованием средств JDK 1.4.

Модифицированный цикл for

Модифицированный цикл `for` (его часто называют циклом "for each") должен быть преобразован в обычный цикл.

5.0	1.4

`for (`*тип переменная*`:` *массив*`)`
`{`
тело цикла
`}`
	`for (int i = 0; i <` *массив*`.length; i++)`
	`{`
	тип переменная `=` *массив*`[i];`
	тело цикла
	`}`

`for (`*тип переменная*`:` *списочный_массив*`)`
`{`
тело цикла
`}`
	`for (int i = 0; i <` *списочный_массив*`.size(); i++)`
	`{`
	тип переменная `=` *(тип)* *списочный_массив*`.get(i);`
	тело цикла
	`}`

Универсальный списочный массив

При использовании универсального списочного массива надо удалить параметры типа и для каждой операции `get()` использовать приведение типа.

5.0	1.4
`ArrayList<`*тип*`>` *списочный_массив* `=` `new ArrayList<`*тип*`>()`	`ArrayList` *списочный_массив* `= new ArrayList()`
списочный_массив`.get(i)`	*(тип)* *списочный_массив*`.get(i)`

Преобразование простых типов

До появления JDK 5.0 преобразование простых типов в интерфейсные классы и обратно выполнялось вручную. Для того чтобы создать интерфейсный объект на базе простого типа, включите в программу вызов конструктора.

5.0	1.4
`Integer wrapper = n;`	`Integer wrapper = new Integer(n);`

Преобразовать интерфейсный объект в значение простого типа позволяют методы `intValue()`, `doubleValue()` и т.д.

5.0	1.4
`int n = wrapper;`	`int n = wrapper.intValue();`

Произвольное число параметров

Параметры надо объединить в массив.

5.0	1.4
метод(*прочие_параметры, p1, p2, p3*)	*метод*(*прочие_параметры*, new *тип*[] { *p1, p2, p3* })

Ковариантные возвращаемые типы

До появления JDK 5.0 при переопределении метода нельзя было изменить тип возвращаемого значения. Теперь это ограничение снято. Переопределяемый метод может возвращать значение, являющееся подтипом возвращаемого значения исходного метода.

5.0	1.4
`public Employee clone() { ... }`	`public Object clone() { ... }`
`...`	`...`
`Employee cloned = e.clone();`	`Employee cloned = (Employee) e.clone();`

Статическое импортирование

В версиях JDK, предшествующих 5.0, статическое импортирование не поддерживалось. Поэтому при вызове статических методов и указании констант надо задавать имя класса.

5.0	1.4
`import static java.lang.Math;`	`System.out.println(Math.sqrt(Math.PI));`
`import static java.lang.System;`	
`...`	
`out.println(sqrt(PI));`	

Консольный ввод

До появления JDK 5.0 класса `Scanner` *не было. Вместо него надо использовать* метод `JOptionPane.showInputDialog()`.

5.0	1.4

`Scanner in = new Scanner(System.in);`
`System.out.print(prompt);`
`int n = in.nextInt();`
`double x = in.nextDouble();`
`String s = in.nextLine();`
	`String input =`
	`  JOptionPane.showInputDialog(prompt);`
	`int n = Integer.parseInt(input);`
	`double x = Double.parseDouble(input);`
	`s = input;`

Форматированный вывод

Метод `printf()` был реализован только в JDK 5.0. При работе с более ранними версиями следует использовать метод `NumberFormat.getNumberInstance()`.

5.0	1.4

`System.out.printf("%8.2f", x);`	`NumberFormat formatter =`
	`  NumberFormat.getNumberInstance();`
	`formatter.setMinimumFractionDigits(2);`
	`formatter.setMaximumFractionDigits(2);`
	`String formatted =`
	`  formatter.format(x);`
	`for (int i = formatted.length();`
	`  i < 8; i++) System.out.print(" ");`
	`System.out.print(formatted);`

Делегирование вызовов панели содержимого

До появления JDK 5.0 классы `JFrame`, `JDialog` и `JApplet` не делегировали вызовы методов `add()` и `setLayout()` панели содержимого. Заметьте, что эта особенность проявляется только на этапе выполнения программы.

5.0	1.4
`add(component)`	`getContentPane().add(component)`
`setLayout(manager)`	`getContentPane().setLayout(manager)`

Кодовые точки Unicode

JDK 5.0 поддерживает Unicode 4.0, где дополнительные символы кодируются двумя последовательно расположенными значениями типа `char`.

5.0	1.4
`int cp = string.codePointAt(i)`	`char cp = string.charAt(i)`
`if (Character.isSupplementaryCharacter(cp))` `...`	Версии, предшествующие JDK 5.0, не поддерживают дополнительные символы

Класс StringBuilder

В JDK 5.0 был введен класс `StringBuilder`, методы которого не синхронизированы. Это несколько повышает эффективность работы данного класса по сравнению с `StringBuffer`.

5.0	1.4
`StringBuilder`	`StringBuffer`

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: info@williamspublishing.com
WWW: http://www.williamspublishing.com

Информация для писем из:

России: 115419, Москва, а/я 783
Украины: 03150, Киев, а/я 152

Предметный указатель

A

Abstract Window Toolkit, 306
AWT, 306

E

Eclipse, 42; 46
Emacs, 42; 49

F

FTP, 24

G

GMT, 138
Graphics User Interface, 305
Greenwich Mean Time, 138
GUI, 305
GZIP, 757

H

HTML, 584
HTTP, 24
HyperText Markup Language, 584

I

IANA Character Set Registry, 747
IFC, 306
Internet Foundation Classes, 306
IP, 24

J

J++, 28
J2EE, 24; 38
J2ME, 38
J2SE, 38
JAR-файл, 181
Java 2 Enterprise Edition, 24; 38
Java 2 Micro Edition, 38
Java 2 Standard Edition, 38
Java Development Kit, 37
Java Network Launch Protocol, 629
Java Web Start, 583; 628
Java Web Start, 628
Java 2D, 323
JBuilder, 42
JDB, 720
JDK, 37
JEdit, 42; 50
JIT, 27
JNLP, 629; 631
Just-in-time compiler, 27; 206

N

NetBeans, 38; 42

P

Proxy-класс, 298

S

Secure Hash Algorithm, 784
SHA, 784
Sun Java Studio, 38
Sun ONE Studio, 38
Swing, 306; 409
Synth, 308

T

TCP, 24
TextPad, 42; 50

U

UML, 132
Unicode, 66
Unified Modeling Language, 132
Universal Coordinated Time, 138
URL, 24
UTC, 138
UTF-16, 66

X

XML, 651

Z

ZIP, 757
ZIP-файл, 757

А

Абсолютное позиционирование, 533
Абсолютный URL, 607
Абстрактный класс, 208
Автоматическая генерация документации, 63
Активное окно, 388; 389
Алгоритм SHA, 784
Анонимный
  внутренний класс, 283; 293
  массив, 114
Аплет, 28; 54
Арифметический оператор, 70
Архив, 181
Архитектура, 410
Атрибут, 596
  align, 597; 601
  archive, 599
  classid, 601
  code, 596; 598
  codebase, 599
  codetype, 601
  height, 596; 597; 601
  name, 599
  object, 599
  width, 596
  widtht, 597

Б

Базовая линия, 339
Базовый
  каталог, 179
  класс, 194
Безымянный внутренний класс, 283; 293
Блок, 61; 94
  catch, 665
  finally, 669
  try, 665
Блокирование пунктов меню, 496
Блокировка файлов, 819
Браузер
  HotJava, 31
  Mosaic, 31

В

Ввод
  информации, 86
  целых чисел, 435
Верификатор, 439
Весовое поле, 514
Виртуальная машина Java, 30; 584
Виртуальная память, 811
Виртуальный код клавиши, 374
Владелец фокуса, 389
Вложенные блоки, 94
Внешнее заполнение, 515
Внутреннее заполнение, 515
Внутренний класс, 265; 282
  в роли обработчика события, 357
Вывод информации, 86
Вызов
  по значению, 161
  по имени, 161
  по ссылке, 161
Высота шрифта, 339
Выход за пределы массива, 660

Г

Генерация исключений, 663
Герметизация пакета, 185; 628
Гравировка, 463
Граница буфера, 817
Графический пользовательский интерфейс, 305
Графическое приложение, 52
Группа, 822
  переключателей, 458

Д

Действие, 392
Декрементирование, 71
Дескриптор, 596
  <applet>, 596; 600
  <object>, 589; 601

<param>, 601
<title>, 584
Диагностическое утверждение, 699
Диаграмма классов, 132
Диалоговое окно, 540
для выбора файлов, 564
для выбора цвета, 576
Динамически расширяемая программа, 204
Динамический компилятор, 206
Динамическое связывание, 202; 203
Диспетчер
защиты, 592
компоновки, 417
BorderLayout, 418; 419
BoxLayout, 507; 508
FlowLayout, 417
GridBagLayout, 507; 512
GridLayout, 421
OverlayLayout, 508
SpringLayout, 524
регистрации, 685
Документирование, 185
Дополнительная плоскость, 66
Дополнительный символ, 66
Дочерний класс, 194

Ж

Жизненный цикл аплета, 591

З

Загрузчик класса, 620; 701
Закладка, 611
Запись
байтов, 730
текста в поток вывода, 753
Заполнение фигур, 334
Заполнитель, 509
Запрос типа, 847
Запуск программ из текстового редактора, 49
Значение NaN, 65

И

Иерархия
наследования, 200
событий AWT, 369
Импортирование, 176
статических методов, 177
статических полей, 177
Имя класса, 60
Инициализационный блок, 171
Инициализация
массива, 114
переменной, 69
поля по умолчанию, 167
Инкапсуляция, 129
Инкрементирование, 71
Инкрементный регулятор, 477
Инсталляция
JDK, 37
библиотек, 40
документации, 40
Интегрированная среда разработки, 46
Интерактивная справочная система, 84
Интерлиньяж, 339
Интерпретатор Java, 26; 44
Интерфейс, 266
Action, 393; 399
ActionListener, 353; 366; 370; 372; 393
AdjustmentListener, 370; 372
Appendable, 735; 736
AppletStub, 617
BasicService, 633; 641
ButtonModel, 414; 462
Cloneable, 276
Closeable, 735; 736
Comparable, 266; 270
DataInput, 742
DataOutput, 743
DateOutput, 741
Document, 434
DocumentListener, 434
Externalizable, 785
FileNameFilter, 806
Filter, 690; 699
Flushable, 735; 736
FocusListener, 370; 372
InvocationHandler, 299; 304
ItemListener, 370; 372
KeyListener, 370; 372
LayoutManager, 538
MenuListener, 497

MouseListener, 370; 372
MouseMotionListener, 370; 372
MouseWheelListener, 370; 372
PropertyChangeListener, 408
Readable, 735; 736
Serializable, 780; 785
Shape, 323
WindowFocusListener, 370; 372; 392
WindowListener, 365; 368; 370; 372
WindowStateListener, 370; 372
обработчика, 373
Интерфейс-маркер, 276
Интерфейсный класс, 234
Исключение, 241; 659
Исключительная блокировка, 820
Источник события, 350; 360; 373

К

Канал, 811
Карта свойств, 642
Клавиша быстрого доступа, 493
Клавишное сокращение, 493
Класс, 60; 129
AbstractAction, 490
AbstractButton, 372; 462; 489; 490; 491; 495
AbstractFormatter, 449
AbstractSpinnerModel, 485
AccessibleObject, 252
ActionEvent, 350; 361; 370; 371; 372
AdjustmentEvent, 370; 371; 372
Applet, 585; 591; 592; 606; 608; 611
AppletContext, 611
Array, 255
ArrayList, 226; 229
ArrayList<T>, 232
Arrays, 117; 119; 270
AWTEvent, 369
BigDecimal, 110; 112
BigInteger, 110; 111
Boolean, 234
BorderFactory, 465
BorderLayout, 419
Box, 512
Buffer, 815; 818
BufferedInputStream, 740
BufferedOutputStream, 740
ButtonGroup, 462
Byte, 234
ByteBuffer, 753; 816
Character, 234
CharBuffer, 736; 753; 818
CharSequence, 736
Charset, 747; 753
Class, 226; 239; 242; 246; 252; 627
ClassLoader, 703
Color, 331; 332
Component, 316; 323; 334; 348; 372; 388; 391; 427; 512; 534
ComponentEvent, 373
ConsoleHandler, 690; 697
Constructor, 242; 243; 247
Container, 356; 417; 419
DataInputStream, 737
DateEditor, 485
DateFormat, 449
DefaultFormatter, 450
DocumentEvent, 434
DocumentFilter, 450
Double, 234
Ellipse2D, 326
Enum, 260; 261
Error, 660
EventHandler, 361
EventListenerList, 407
EventObject, 350; 360; 369
EventObject, 360
EventTracer, 712
Exception, 660
Field, 243; 247; 252
File, 804; 807
FileChannel, 812; 819; 820
FileContents, 641
FileFilter, 575
FileHandler, 690; 697
FileInputStream, 737; 739; 815
FileLock, 821
FilenameFilter, 810
FileOpenService, 633; 641
FileOutputStream, 737; 740; 815
FileSaveService, 642
FileView, 576
FilterInputStream, 738
FilterOutputStream, 738

Float, 234
FocusAdapter, 370
FocusEvent, 370; 371; 372; 392
Font, 342
Formatter, 699
Frame, 309; 317
GenericArrayType, 866
Graphics, 323; 333; 347
Graphics2D, 323; 334; 343
GraphicsDevice, 315
GraphicsEnvironment, 315; 336; 719
GregorianCalendar, 143
GridConstraints, 523
GridLayout, 425
Hashtable, 644
Image, 347
ImageIcon, 356
ImageIO, 347
InputEvent, 379; 387
InputStream, 730; 731; 734
InputStreamReader, 746
Integer, 234; 236
ItemEvent, 370; 371; 372
JButton, 356; 414; 563
JCheckBox, 458
JCheckBoxMenuItem, 491
JColorChooser, 576; 582
JComboBox, 372; 470
JComponent, 323; 400; 427; 433; 467; 505; 563
JDialog, 557
JFileChooser, 564; 574
JFormattedTextField, 448
JFrame, 309; 322; 489
JLabel, 428
JMenu, 399; 487
JMenuItem, 488; 490; 495; 497
JOptionPane, 89; 282; 550
JPanel, 320; 421
JPasswordField, 434
JPopupMenu, 493
JRadioButton, 462
JRadioButtonMenuItem, 491
JRootPane, 563
JScrollbar, 372
JScrollPane, 455
JSlider, 476
JSpinner, 477; 484
JTextArea, 425; 454
JTextComponent, 425; 426
JTextField, 372; 414; 425; 427
JToolBar, 505
KeyAdapter, 370
KeyboardFocusManager, 391
KeyEvent, 370; 371; 379
KeyStroke, 399
LineBorder, 467
LineMetrics, 343
Logger, 695
LogRecord, 698
Long, 234
LookAndFeelInfo, 364
MappedByteBuffer, 812
MaskFormatter, 451
Matcher, 830
Math, 74; 134
MediaTracker, 348
Method, 243; 247; 259; 865
Modifier, 247
MouseAdapter, 370
MouseEvent, 370; 371; 372; 387; 493
MouseMotionAdapter, 370
MouseWheelEvent, 370; 371; 372
NumberFormat, 237
Object, 214; 226
ObjectInputStream, 783
ObjectOutputStream, 779; 782
OutputStream, 730; 732; 734
OutputStreamWriter, 746
ParameterizedType, 866
Pattern, 829
PersistenceService, 642
Preferences, 650; 655
PrinterWriter, 753
PrintWriter, 755; 766
Properties, 643; 646
PropertyChangeEvent, 408
Proxy, 304
PushbackInputStream, 738; 741
Random, 175
RandomAccessFile, 744; 745; 773; 815
Reader, 734
Rectangle2D, 326
RectangularShape, 326; 330

Robot, 716; 719
RuntimeException, 660; 674
Scanner, 86; 89; 756
SerialCloneable, 802
ServiceManager, 641
Short, 234
SimpleDateFormat, 485
SoftBevelBorder, 466
SpinnerListMode, 484
SpinnerNumberModel, 484
Spring, 533
SpringLayout, 532
StackTraceElement, 674
String, 79; 82
StringBuffer, 772
StringBuilder, 772
StringTokenizer, 767
SwingUtilities, 563
System, 89; 119; 649
SystemColor, 332
Throwable, 242; 659; 665; 673
Timer, 282; 372
Toolkit, 282; 318; 388
TypeVariable, 866
UIManager, 364
URL, 607; 608
Void, 234
WildcardType, 866
Window, 317; 372; 391; 425
WindowAdapter, 366; 370
WindowEvent, 350; 368; 370; 371; 372; 392
Writer, 734
ZipEntry, 764
ZipFile, 765
ZipInputStream, 735; 763
ZipOutputStream, 735; 763; 764
ZipUnputStream, 757
контроллера, 410
модели, 410
представления, 410
символов, 821
Класс-адаптер, 366
Клон, 265; 275
Клонирование, 265; 274; 802
Ключевое слово
abstract, 867
assert, 700; 867
boolean, 867
break, 867
byte, 867
case, 867
catch, 867
char, 867
class, 60; 867
const, 867
continue, 867
default, 867
do, 867
double, 867
else, 867
extends, 194; 867
final, 69; 155; 867
finally, 867
float, 867
for, 867
goto, 867
if, 867
implements, 267; 867
import, 176; 867
int, 867
interface, 868
long, 868
native, 868
new, 868
null, 868
package, 178; 868
private, 868
protected, 214; 868
public, 60; 868
return, 868
short, 868
static, 70; 155; 868
strictfp, 71; 868
super, 196; 868
switch, 868
synchronized, 868
this, 868
throw, 868
throws, 868
transient, 868
try, 868
unstanceOf, 867
void, 868

volatile, 868
while, 868
Кодовая
единица, 78
плоскость, 66
точка, 66; 78
Командная строка, 116
Комментарии, 63
к классу, 186
к методу, 186
к пакету, 189
к полю, 187
общего характера, 187
Компиляция программ из текстового редактора, 49
Компоновка, 415; 417
Конкатенация строк, 80
Консольное окно, 60; 710
Константа, 69; 156
класса, 70
Конструктор, 135
по умолчанию, 168
Контекст аплета, 609
Контекстное меню, 492
Контролируемое исключение, 240; 241; 660; 661; 662
Контроллер, 410
Копирование массивов, 115
Корневая область, 318

Л

Лексема, 767
Линейный регулятор, 471
Логический оператор, 72
Логическое
значение, 67
название шрифта, 337
Локальный внутренний класс, 290

М

Максимальная
высота, 339
глубина посадки, 339
Маркер буфера, 817
Маска, 73
Маскировка, 170
Массив, 112
Математическая функция, 74
Матовая рамка, 463
Меню, 486
Метка, 428
case, 106
Метод, 129
accept, 575; 810
actionPerformed, 372; 393
add, 111; 112; 144; 229; 323; 356; 399; 407; 419; 462; 487; 505
addDocumentListener, 434
addHandler, 697
addImage, 348
addItem, 470
addLayoutComponent, 538
addPropertyChangeListener, 393
addSeparator, 488; 505
adjustmentValueChanged, 372
aliases, 753
append, 455; 736; 772
appendCodePoint, 772
array, 753
arraycopy, 119
available, 732
availableCharsets, 753
beep, 282
binarySearch, 119
canRead, 641; 807
canWrite, 641; 807
capacity, 818
cast, 860
changedUpdate, 434
charAt, 82; 736; 753
checkError, 756
clear, 818
clearAssertionStatus, 703
clone, 226
close, 697; 732; 736
closeEntry, 764
codePointAt, 82
codePointCount, 83
commitEdit, 449
compareTo, 82; 112; 270
compile, 829
constant, 533
copyArea, 348
countTokens, 768

create, 361; 642
createBevelBorder, 466
createCompoundBorder, 466
createCustomCursor, 388
createDialog, 582
createEmptyBorder, 465
createEtchedBorder, 465
createGlue, 512
createHorizontalBox, 512
createHorizontalGlue, 512
createHorizontalStrut, 512
createImage, 348
createLineBorder, 465
createLoveredBevelBorder, 466
createMatteBorder, 465
createNewFile, 810
createRigidArea, 512
createTempFile, 807
createTitledBorder, 466
createVerticalBox, 512
createVerticalGlue, 512
createVerticalStrut, 512
decode, 753
delay, 719
delete, 642; 773; 808
deleteOnExit, 808
deriveFont, 342
destroy, 591; 592
dispose, 312; 348
divide, 111; 112
drawImage, 347
drawString, 343
encode, 753
end, 830
endsWith, 82
ensureCapacity, 229
entries, 765
equals, 82; 226
equals(), 215
equalsIgnoreCase, 82
exists, 808
exit, 89
exportNode, 656
exportSubtree, 655
fill, 120
finalize(), 175
find, 830
flip, 818
flush, 347; 697; 732; 736
focusGained, 372
focusLost, 372
format, 699
forName, 242; 753
get, 143; 233; 252; 255; 642; 655; 816; 818
getAbsolutePath, 808
getActionCommand, 361; 372; 415; 462
getActionMap, 400
getActionSource, 370
getActiveWindow, 391
getActualTypeArguments, 866
getAdjustable, 372
getAdjustmentType, 372
getAlignmentX, 512
getAlignmentY, 512
getAncestorOfClass, 563
getApplet, 611
getAppletContext, 611
getAppletInfo, 606
getApplets, 611
getAscent, 343
getAudioClip, 608
getBoolean, 655
getBounds, 866
getByteArray, 655
getCanonicalFile, 808
getCanonicalPath, 808
getCause, 673
getCenterX, 330
getCenterY, 330
getChannel, 815
getClass, 226
getClass(), 216
getClassName, 364; 674
getClickCount, 372; 387
getCodeBase, 608; 641
getColor, 582
getComponent, 373
getComponentPopupMenu, 493
getConstraint, 532
getConstraints, 532
getConstructor, 861
getConstructors, 246
getContentPane, 322
getCrc, 764

getCurrentKeyboardFocusManager, 391
getDeclaredConstructor, 861
getDeclaredConstructors, 246
getDeclaredField, 252
getDeclaredFields, 246; 252
getDeclaringClass, 247
getDefaultScreenDevice, 719
getDefaultToolkit, 282; 318
getDescent, 343
getDescription, 575; 576
getDocument, 434
getDocumentBase, 608
getDocumentFilter, 449
getDouble, 655
getEntry, 765
getEnumConstants, 860
getExceptionTypes, 247
getExtendedState, 317
getFamily, 342
getField, 252
getFields, 246; 252
getFileName, 674
getFilePointer, 745
getFilter, 697
getFocusLostBehaviour, 449
getFocusowner, 391
getFocusWindow, 391
getFontName, 342
getFontRenderContext, 343
getFormatter, 697
getGenericComponentType, 866
getGenericInterfaces, 865
getGenericParameterTypes, 865
getGenericReturnType, 865
getGenericSuperclass, 865
getGraphics, 347
getHandlers, 696
getHead, 699
getHeight, 330; 343; 532
getIcon, 576
getImage, 318; 609
getInheritsPopupMenu, 493
getInputMap, 400
getInputStream, 641; 765
getInt, 655
getInvalidCharacters, 451
getItem, 372
getItemSelectable, 372
getKeyChar, 372; 379
getKeyCode, 372; 379
getKeyModifiersText, 372; 379
getKeyStroke, 399
getKeyText, 372; 379
getLeading, 343
getLength, 256; 434
getlevel, 697; 698
getLineMetrics, 342
getLineNumber, 674
getListenerList, 408
getListeners, 408
getLocalGraphicsEnvironment, 719
getLocation, 316
getLocationOnScreen, 316
getLogger, 695
getLoggerName, 698
getLong, 655
getLowerBounds, 866
getMaximumValue, 533
getMaxX, 330
getMaxY, 330
getMessage, 665; 698
getMethodName, 674
getMethods, 246
getMillis, 698
getMinimumValue, 533
getMinX, 330
getMinY, 330
getMnemonic, 415
getModifiers, 247; 372; 379
getModifiersEx, 387
getModifiersExText, 388
getName, 226; 247; 342; 364; 576; 641; 764; 765; 808; 866
getNames, 642
getNewState, 368; 372
getNextEntry, 763
getNextValue, 485
getOldState, 368; 372
getOppositeComponent, 392
getOppositeWindow, 372
getOutputStream, 641
getOverwriteMode, 450
getOwnerType, 866
getParameter, 606

getParameterInfo, 606
getParameters, 698
getParameterTypes, 247
getParent, 696; 808
getParentFile, 808
getPassword, 435
getPath, 809
getPlaceholder, 451
getPlaceholderCharacter, 451
getPoint, 372; 387
getPreferred, 533
getPreviousValue, 485
getProperties, 649
getProperty, 647; 649
getProxy, 304
getRawType, 866
getResource, 627
getResourceAsStream, 627
getResourceBundle, 698
getResourceBundleName, 698
getReturnType, 247
getRootPane, 563
getScreenSize, 318
getScrollAmount, 372
getSelectedFile, 575
getSelectedFiles, 575
getSelectedItem, 470
getSelection, 462
getSequenceNumber, 699
getServiceNames, 641
getSize, 764
getSource, 360; 370
getSourceClassName, 698
getSourceMethodName, 698
getStackTrace, 673
getStateChange, 372
getStringBounds, 342
getSuperclass, 226; 861
getTail, 699
getText, 426; 434
getThreadID, 699
getThrown, 698
getTypeDescription, 576
getTypeParameters, 865
getUpperBounds, 866
getUseParentHandlers, 697
getValidCharacters, 451
getValue, 372; 393; 399; 448; 484; 485; 533
getValueContainsLiteralCharacters, 451
getWheelRotation, 372
getWidth, 330; 532
getWindow, 372
getX, 330; 372; 387; 532
getY, 330; 372; 387; 532
group, 830
groupCount, 830
hashCode, 219
hasMoreTokens, 768
hasNext, 89
hasNextDouble, 89
hasNextInt, 89
hasRemaining, 815
height, 533
importPreferences, 656
indexOf, 82
init, 591
initCause, 673
insert, 488; 773
insertSeparator, 488
insertString, 450
insertUpdate, 434
intValue, 236
invoke, 259; 304
isAbstract, 247
isAccessible, 252
isActionKey, 372; 379
isActive, 392
isAltDown, 380
isAltGraphDown, 380
isArmed, 415
isControlDown, 380
isDefaultButton, 563
isDirectory, 764; 809
isEditValid, 449
isEnabled, 316; 393; 399; 415
isFile, 809
isFinal, 247
isFocusable, 391
isFocused, 391
isFocusOwner, 391
isHidden, 809
isInterface, 247
isMetaDown, 380
isNative, 247

isNativeMethod, 674
isPopupTrigger, 493
isPressed, 415
isPrivate, 247
isProtected, 247
isProxyClass, 304
isPublic, 247
isRollover, 415
isSelected, 415; 458; 491
isShiftDown, 380
isShowing, 316
isStatic, 247
isStrict, 247
isSynchronized, 248
isTemporary, 372
isTraversable, 576
isUndecorated, 317
isVisible, 316
isVolatile, 248
isWebBrowserSupported, 641
itemStateChanged, 372
keyPress, 719
keyPressed, 372
keyRelease, 719
keyReleased, 372
keys, 655
keyTyped, 372
lastIndexOf, 83
lastModified, 809
layoputContainer, 538
length, 83; 737; 745; 772; 809
limit, 816
list, 809
listFiles, 809
listRoots, 810
load, 647
lock, 820
lookingAt, 830
lookup, 641
main(), 61
map, 815
mark, 732; 818
markSupported, 732
matcher, 829
matches, 830
menuCanceled, 497
menuDeselected, 497
menuSelected, 497
minimumLayoutSize, 538
minus, 533
mkdir, 810
mkdirs, 810
mod, 111
mouseClicked, 372
mouseDragged, 372
mouseEntered, 372
mouseExited, 372
mouseMove, 719
mouseMoved, 372
mousePress, 719
mousePressed, 372
mouseRelease, 719
mouseReleased, 372
mouseWheelMoved, 372
multiply, 111; 112
newInstance, 242; 256; 860
newProxyInstance, 304
next, 89
nextDouble, 89
nextInt, 89; 175
nextLine, 89
nextToken, 768
node, 655
offsetByCodePoints, 82
openFileDialog, 641
openMultiDialog, 642
order, 817
pack, 425
paintComponent, 323
parse, 237
parseInt, 237
pitValue, 393
play, 608
position, 818
preferredLayoutSize, 538
print, 755
printf, 90
println, 756
printStackTrace, 242
propertyChange, 408
publish, 697
put, 655; 816; 818
putBoolean, 655
putByteArray, 655

putConstraints, 532
putDouble, 655
putFloat, 655
putInt, 655
putLong, 655
putNextEntry, 764
putValue, 399
read, 347; 731; 736
readBoolean, 742
readByte, 743
readChar, 743
readDouble, 743
readFloat, 743
readFully, 743
readInt, 743
readLine, 743
readLong, 743
readObject, 783
readShort, 743
readUTF, 743
release, 821
remaining, 818
remove, 233; 408; 450; 488
removeAllItems, 470
removedUpdate, 434
removeHandler, 697
removeItem, 470
removeLayoutComponent, 538
removePropertyChangeListener, 393
renameTo, 810
repaint, 323
replace, 83; 450
replaceAll, 830
replaceFirst, 830
requestFocus, 391
requestFocusInWindow, 391
reset, 732; 818; 831
resetChoosableFileFilters, 575
resize, 591
revalidate, 427
rewind, 818
saveFileDialog, 642
seek, 745
set, 144; 232; 252; 255
setAcceptAllFileFilterUsed, 575
setAccessible, 252
setAccessory, 575
setAction, 489
setActionCommand, 462
setBackground, 334
setBorder, 467
setCharAt, 773
setClassAssertionStatus, 703
setColor, 333; 582
setColumns, 427; 455
setComponentPopupMenu, 493
setConstarint, 532
setCrc, 765
setCurrentDirectory, 574
setCursor, 388
setDefaultAssertionStatus, 703
setDisplayedMnemonicIndex, 495
setEchoChar, 435
setEditable, 426
setEditor, 484
setEnabled, 393; 399; 497
setExtendedState, 317
setFicusLostBehaviour, 449
setFileFilter, 575
setFileSelectionMode, 574
setFileView, 575
setFilter, 697
setFocusable, 391
setFont, 343
setForeground, 334
setFormatter, 697
setHorizontalTextPosition, 490
setIcon, 429
setInheritsPopupMenu, 493
setInvalidCharacters, 451
setJMenuBar, 489
setlabelTable, 477
setLastModified, 810
setLayout, 417
setLevel, 697; 764
setLineWrap, 455
setLookAndFeel, 364
setMajorTickSpacing, 477
setMethod, 764; 765
setMinorTickSpacing, 477
setMnemonic, 495
setMultiSelectionEnabled, 574
setOverwriteMode, 450
setPackageAssertionStatus, 703

setPaintLabels, 477
setPaints, 334
setPaintTicks, 477
setPaintTrack, 477
setParent, 696
setPlaceholder, 451
setPlaceholderCharacter, 451
setPreferredSize, 433
setReadOnly, 810
setResizable, 317
setRows, 455
setSelected, 458; 491
setSelectedFile, 574
setSelectedFiles, 574
setSize, 765
setSnapToTicks, 477
setTabSize, 455
setText, 426; 429
setTime, 144
setTitle, 317
setToolTipText, 505
setUndecorated, 317
setUseParentHandlers, 697
setValidCharacter, 451
setValue, 449; 484; 485; 533
setValueContainsLiteralCharacters, 451
setVisible, 316
setWrapStyleForward, 455
showConfirmDialog, 541; 550
showDialog, 575; 582
showDocument, 611; 641
showInputDialog, 89; 541; 552
showInternalConfirmDialog, 550
showInternalInputDialog, 552
showInternalMessageDialog, 550
showInternalOptionDialog, 551
showMessageDialog, 282; 541; 550
showOpenDialog, 575
showOptionDialog, 541; 551
showSaveDialog, 575
showStatus, 611
showStatus(), 610
size, 229
skip, 732
skipBytes, 743
sort, 119; 270
split, 829
start, 282; 591; 592; 830
startWith, 83
stop, 282; 591; 592
store, 647
stringToValue, 449
subSequence, 737
substring, 83
subtract, 111; 112
sum, 533
systemNodeForPackage, 655
systemRoot, 655
toBack, 317
toFront, 317
toLowerCase, 83
toPattern, 485
toString, 221; 226; 237; 247; 261; 674; 737; 753; 773
toUpperCase, 83
translatePoint, 372; 387
trim, 83
trimToSize, 229
tryLock, 820
unread, 741
userNodeForPackage, 655
userRoot, 655
validate, 427
valueOf, 112; 237; 261
valueToString, 449
waitForAll, 348
waitForID, 348
width, 533
windowActivated, 368; 372
windowClosed, 368; 372
windowClosing, 368; 372
windowDeactivated, 368; 372
windowDeiconified, 368; 372
windowGainedFocus, 372; 392
windowIconified, 368; 372
windowLostFocus, 372; 392
windowOpened, 368; 372
windowStateChanged, 368; 372
wrap, 753
write, 732
writeBoolean, 741; 743
writeByte, 741; 744
writeChar, 741; 744
writeChars, 741; 744

writeDouble, 741; 744
writeFloat, 741; 744
writeInt, 741; 744
writeLong, 741; 744
writeObject, 783
writeShort, 741; 744
writeUTF, 741; 744
Многоадресная передача событий, 400
Многомерный массив, 120
Множественное наследование, 273
Модальное диалоговое окно, 540
Модель, 410
делегирования событий, 350
Модификатор доступа, 60
Модифицирующий метод, 140
Модуль, 134

Н

Наблюдатель, 413
Набор символов, 746
Big5, 749
Big5-HKSCS, 749
Cp037, 749
Cp1006, 751
Cp1025, 751
Cp1026, 751
Cp1046, 751
Cp1097, 751
Cp1098, 751
Cp1112, 751
Cp1122, 751
Cp1123, 751
Cp1124, 751
Cp1140, 751
Cp1141, 751
Cp1142, 751
Cp1143, 751
Cp1144, 751
Cp1145, 751
Cp1146, 751
Cp1147, 751
Cp1148, 751
Cp1149, 752
Cp1381, 752
Cp1383, 752
Cp273, 749
Cp277, 749
Cp278, 749
Cp280, 749
Cp284, 750
Cp285, 750
Cp297, 750
Cp33722, 752
Cp420, 750
Cp424, 750
Cp437, 750
Cp500, 750
Cp737, 750
Cp775, 750
Cp838, 750
Cp850, 750
Cp852, 750
Cp855, 750
Cp856, 750
Cp857, 750
Cp858, 750
Cp860, 750
Cp861, 750
Cp862, 750
Cp863, 750
Cp864, 750
Cp865, 750
Cp866, 750
Cp868, 750
Cp869, 750
Cp870, 750
Cp871, 750
Cp874, 750
Cp875, 750
Cp918, 750
Cp921, 750
Cp922, 750
Cp930, 750
Cp933, 751
Cp935, 751
Cp937, 751
Cp939, 751
Cp942, 751
Cp942C, 751
Cp943, 751
Cp943C, 751
Cp948, 751
Cp949, 751
Cp949C, 751

Cp950, 751
Cp964, 751
Cp970, 751
EUC-JP, 749
EUC-KR, 749
GB18030, 749
GBK, 749
ISCII91, 749
ISO2022CN, 752
ISO2022CN_CNS, 752
ISO2022CN_GB, 752
ISO-2022-JP, 749
ISO-2022-KR, 749
ISO-8859-1, 748
ISO8859-13, 748
ISO8859-15, 748
ISO8859-2, 748
ISO8859-3, 749
ISO8859-4, 748
ISO8859-5, 748
ISO8859-6, 749
ISO8859-7, 748
ISO8859-8, 749
ISO8859-9, 748
JIS0201, 752
JIS0208, 752
JIS0212, 752
JISAutoDetect, 752
MacArabic, 752
MacCentralEurope, 752
MacCirilli, 752
MacGreek, 752
MacHebrew, 752
MacIceland, 752
MacRomania, 752
MacThai, 752
MacTurkish, 752
MacUkraine, 752
MS874, 752
Shift_JIS, 749
TIS-620, 749
US-ASCII, 748
UTF-16, 748
UTF-16BE, 748
UTF-16LE, 748
UTF-8, 748
windows-1250, 748
windows-1251, 748
windows-1252, 748
windows-1253, 748
windows-1254, 748
windows-1255, 749
windows-1256, 749
windows-1257, 748
windows-1258, 749
windows-31j, 749
windows-950, 749
x-EUC-CN, 749
x-EUC-JP-LINUX, 749
x-EUC-TW, 749
x-MS950-HKSCS, 749
x-mswin-936, 749
x-windows-949, 749
Наследование, 130; 131; 193
Начертание символов, 336
Неизменяемый объект, 79
Неконтролируемое исключение, 241; 660; 662
Немодальное диалоговое окно, 540
Немодифицируемый объект, 79
Непротиворечивость, 217
Неявный параметр, 151
Низкоуровневое событие, 371; 374
Низкоуровневый код, 841
Нумерованный тип, 77; 260

О

Обзор, 189
Область
  слоя, 318
  содержимого, 318
Обработка событий, 349
Обработчик
  вызовов, 299
  исключений, 240; 241; 659
  событий, 350; 373
Обратный вызов, 279
Объект, 129
  события, 350
Объектная переменная, 136
Объектно-ориентированное программирование, 127
Объектный поток, 778
Объем буфера, 817

Оконный диспетчер, 388
Округление, 76
Оператор, 70
  break, 106
  break с меткой, 108
  instanceof, 207
  switch, 106
  отношения, 72
Основная многоязыковая плоскость, 66
Основной тип, 63
Отладчик среды Eclipse, 725
Относительный URL, 607
Отношение
  агрегирования, 131
  зависимости, 131
  наследования, 131; 132
Отображение файла в память, 811
Отпечаток версии, 800
Отражение, 193; 238
Отрицательная бесконечность, 65

П

Пакет, 175
  по умолчанию, 178
  ресурсов, 686
Панель, 318; 419
  инструментов, 500
Параметр, 62; 161
  типа, 834
  командной строки, 116
Перегрузка, 167
Переключатель, 458
Переменная, 67
  окружения CLASSPATH, 181; 183
Переменное число параметров, 237
Переопределение метода, 195
Перехват исключений, 665
Пиктограмма, 489
Платформа Java, 21
Побитовая операция, 73
Поведение объекта, 130
Поверхностное копирование, 276
Подкласс, 194
Подменю, 486
Подсказка, 502
Подстановочный тип, 851
Позиционирование фреймов, 312
Позиция в буфере, 817
Поле
  для ввода пароля, 434
  редактирования, 425; 426
  экземпляра, 129
Полиморфизм, 201
Полиморфные переменные, 201
Положительная бесконечность, 65
Попытка обратиться к объекту по ссылке null, 660
Порождающий метод, 158
Порядок обхода компонентов, 538
Потеря фокуса ввода, 436
Поток, 27; 729
  ввода, 730
  вывода, 730
  данных, 741
  с произвольным доступом, 773
  управления, 94
Предопределенный класс символов, 823
Представление, 410
Предусловие, 702
Преобразование числовых типов, 75
Приведение типов, 206
Приемы отладки, 704
Принудительный вывод, 731
Принцип
  модульности, 134
  подстановки, 201
Приоритет операторов, 76
Приподнятая фаска, 463
Программа
  appletviewer, 54; 588
  jar, 181; 621
  javadoc, 185
Программирование интерфейсов, 265
Прозрачная область, 318
Производный класс, 194
Простой тип, 63; 161
Пункт меню, 486
Пустая рамка, 463

Р

Разблокирование
  диагностических утверждений, 700
  пунктов меню, 496
Размер экрана, 314

Разрешение перегрузки, 167; 202
Рамка, 462
  составленная из линий, 463
Раскрывающийся список, 467
Распорка, 509; 526
Расширение класса, 130
Регистрация
  обработчика события, 350
  ошибок, 682
Регулярное выражение, 821
Редактируемый раскрывающийся
  список, 467
Режим
  автоматического сброса буфера, 754
  округления, 112
  отображения, 812
  произвольного доступа, 744
Рекомендуемый размер, 508
Ресурс, 624
Рефлексивность, 216
Рефлективная программа, 239
Родительский класс, 194

С

Самораспаковывающийся JAR-файл, 623
Сборка мусора, 175
Семантическое событие, 371
Сервлет, 629
Сериализация, 779
  объектов, 779; 789
  перечислений, 798
Сериализованный объект, 599
Символ-заполнитель, 441
Символьный тип, 66
Симметричность, 216
Синхронный компилятор, 27
Склейка, 509
Событие, 349
  связанное с клавиатурой, 374
  связанное с мышью, 380
  связанное с фокусом ввода, 388
  AWT, 712
Совместимость с предыдущей версией, 800
Совместная блокировка, 820
Совместно используемые строки, 79
Совпадение строк, 81
Создание диалоговых окон, 553
Сокрытие данных, 129
Сортировка массива, 117
Составной оператор, 94
Состояние объекта, 129
Списочный массив, 227
Ссылка на объект, 161
Статическая
  константа, 156
  переменная, 155
Статический
  внутренний класс, 283; 295
  инициализационный блок, 172
  контекст, 849
  метод, 74; 157
Статическое поле, 155
  связывание, 203
Строка, 77
Строка меню, 486
Суперкласс, 194
Сущность объекта, 130
Схема
  наблюдателя, 413
  проектирования, 412

Т

Таблица методов, 203
Текстовая область, 425; 451
Текстовый поток, 745
Тело метода, 62
Терминальный
  класс, 205
  метод, 205
Технология
  ActiveX, 594
  Java Web Start, 583; 628
Тип
  массива, 112
  переменной, 67
  с плавающей точкой, 64
Транзитивность, 216
Трассировка стека, 671

У

Указатель файла, 744
Универсальные типы данных, 833
Универсальный
  класс, 836

метод, 838
Уровень регистрации
CONFIG, 683
FINE, 683
FINER, 683
FINEST, 683
INFO, 683
SEVERE, 683
WARNING, 683
Условный оператор, 95
Утилита
Java Plug-In, 584
serialver, 800
Утопленная фаска, 463

Ф

Файл JAR, 620
манифеста, 622
Фактическое значение, 524
Фиксированная область, 509
Фильтр, 437; 690
потока, 737
Фильтрованный поток, 738
Флажок опции, 455
Фокус ввода, 388
Формат
AIFF, 607
AU, 607
GIF, 607
JNLP, 629
JPEG, 607
MIDI, 607
с разделителями, 766
Форматирование выходных данных, 89
даты и времени, 92
Форматированный ввод, 435
Фрейм, 309
Фрейм-владелец, 553

Х

Хэш-код, 219

Ц

Целочисленный тип, 64
Цепочка наследования, 200
Цикл, 113
do/while, 100
for, 102
while, 98

Ч

Число с двойной точностью, 65
Чтение
байтов, 730
информации с консоли, 86
текста из потока ввода, 756

Ш

Шаблон, 833
Шрифт, 336

Э

Экземпляр класса, 129
Эхо-символ, 434

Я

Явная инициализация поля, 169
Явный параметр, 151
Язык Oak, 30

Научно-популярное издание

Кей С. Хорстманн, Гари Корнелл

Java 2. Библиотека профессионала
Том 1. Основы
7-е издание

Литературный редактор	*И.А. Попова*
Верстка	*А.В. Плаксюк*
Художественный редактор	*С.А. Чернокозинский*
Корректоры	*Т.А. Корзун, О.В. Мишутина, В.В. Смоляр, Л.В. Чернокозинская*

Издательский дом "Вильямс".
127055, Москва, ул. Лесная, д. 43, стр. 1.

Подписано в печать 12.12.2006. Формат 70×100/16.
Гарнитура NewBaskerville. Печать офсетная.
Усл. печ. л. 56,0. Уч.-изд. л. 43,8.
Тираж 2000 экз. Заказ № 3790.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., 15.